# SPRING BATCH

# 参考文档(中文版)

高度精简和优化

大数据量批处理架构

# 目錄

# Spring Batch参考文档中文版

原作者：Lucas Ward , Dave Syer , Thomas Risberg , Robert Kasanicky , Dan Garrette , Wayne Lund , Michael Minella , Chris Schaefer , Gunnar Hillert

版本号：3.0.1.RELEASE 查看原文请点击此处。或者 查看HTML单页面版

在线预览版本地址: 在线预览



本翻译文档由Spring Batch参考中文翻译组于2014年7月21日开始翻译，版权由原文版权所有者和翻译组共同所有。参与翻译请发邮件到：kimmking@163.com。

翻译组成员列表如下：

- kimmking(http://blog.csdn.net/kimmking)
- 铁锚(http://blog.csdn.net/renfufei)
- stormhouse(http://stormhouse.github.io/)
- Sean(http://caochengbo.iteye.com)

完成百分比：5/17 ~ 29%

| 章节 | 页数 | 翻译者 | 状态 | 计划完成时间 |
|---|---|---|---|---|
| 01 | - | 铁锚 | done | 2014-08-10 |
| 02 | - | stormhouse | done | 2014-08-02 |
| 03 | - | Sean | done | 2014-08-03 |
| 04 | - | Sean | done | 2014-08-20 |
| 05 | - | stormhouse | done | 2014-08-20 |
| 06 | - | 铁锚 | done | 2015-04-15 |
| 07 | - | 铁锚 | done | 2014-09-10 |
| 08 | - | 翟伟 | done | 2015-04-15 |
| 09 | - | 赵辉亮 | done | 2015-04-16 |
| 10 | - | 赵辉亮 | doing | 2015-07-16 |
| 11 | - | 翟伟 | doing | 2015-05-15 |
| 12 | - | 铁锚 | doing | 2015-06-01 |
| 13 | - | - | - | - |
| 14A | - | 铁锚 | done | 2015-04-21 |
| 15B | - | - | - | - |
| 16C | - | - | - | - |
| 17G | - | 铁锚 | done | 2015-04-25 |

# Spring Batch介绍

在企业领域,有很多应用和系统需要在生产环境中使用批处理来执行大量的业务操作.批处理业务需要自动地对海量数据信息进行各种复杂的业务逻辑处理,同时具备极高的效率,不需要人工干预.执行这种操作通常根据时间事件(如月末统计,通知或信件),或者定期处理那些业务规则超级复杂,数据量非常庞大的业务,(如保险赔款确定,利率调整),也可能是从内部/外部系统抓取到的各种数据, 通常需要格式化、数据校验、并通过事务的方式处理到自己的数据库中. 企业中每天通过批处理执行的事务多达数十亿.

Spring Batch是一个轻量级的综合性批处理框架,可用于开发企业信息系统中那些至关重要的数据批量处理业务. Spring Batch基于POJO 和 Spring框架,相当容易上手使用,让开发者很容易地访问和利用企业级服务. Spring Batch不是调度(scheduling)框架. 因为已经有很多非常好的企业级调度框架,包括商业性质的和开源的, 例如Quartz, Tivoli, Control-M等.它是为了与调度程序一起协作完成任务而设计的,而不是用来取代调度框架的.

Spring Batch提供了大量的,可重用的功能, 这些功能对大数据处理来说是必不可少的, 包括 日志/跟踪(tracing), 事务管理, 任务处理(processing)统计, 任务重启, 忽略(skip), 和资源管理等功能。 此外还提供了许多高级服务和特性, 使之能够通过优化(optimization ) 和分片技术(partitioning techniques)来高效地执行超大型数据集的批处理任务。

Spring Batch是一个具有高可扩展性的框架,简单的批处理,或者复杂的大数据批处理作业都可以通过Spring Batch框架来实现。

# 背景

在开源项目及其相关社区把大部分注意力集中在基于web和SOA基于消息机制的框架中时，基于java的批处理框架却无人问津，尽管在企业IT环境中一直都有这种批处理的需求。但因为缺乏一个标准的、可重用的批处理框架导致在企业客户的IT系统中存在着很多一次编写,一次使用的版本,以及很多不同的内部解决方案。

SpringSource和Accenture（埃森哲)致力于通过合作来改善这种状况。埃森哲在实现批处理架构上有着丰富的产业实践经验,SpringSource有深入的技术开发积累,背靠Spring框架提供的编程模型,意味着两者能够结合成为默契且强大的合作伙伴,创造出高质量的、市场认可的企业级java解决方案,填补这一重要的行业空白。两家公司目前也正着力于开发基于spring的批处理解决方案，为许多客户解决类似的问题。这同时提供了一些有用的额外的细节和以及真实环境的约束,有助于确保解决方案能够被客户用于解决实际的问题。基于这些原因,SpringSource和埃森哲一起合作开发Spring Batch。

埃森哲已经贡献了先前自己的批处理体系结构框架,这个框架基于数十年宝贵的经验并基于最新的软件平台(如COBOL/Mainframe, C++/Unix 及现在非常流行的Java平台)来构建Spring Batch项目, Spring Batch未来将会由开源社区提交者来驱动项目的开发,增强,以及未来的路线图。

埃森哲咨询公司与SpringSource合作的目标是促进软件处理方法、框架和工具的标准化改进，并在创建批处理应用时能够持续影响企业用户。企业和政府机构希望为他们提供标准的、经验证过的解决方案，而他们的企业系统也将受益于Spring Batch。

# 使用场景

典型的批处理程序通常是从数据库、文件或队列中读取大量数据，然后通过某些方法处理数据，最后将处理好格式的数据写回库中。通常SpringBatch工作在离线模式下,不需要用户干预、就能自动进行基本的批处理迭代，进行类似事务方式的处理。批处理是大多数IT项目的一个组成部分，而Spring Batch是唯一能够提供健壮的企业级扩展性的批处理开源框架。

## 业务场景

- 定期提交批处理任务
- 并发批处理：并行执行任务
- 分阶段，企业消息驱动处理
- 高并发批处理任务
- 失败后手动或定时重启
- 按顺序处理任务依赖(使用工作流驱动的批处理插件)
- 局部处理：跳过记录(例如在回滚时)
- 完整的批处理事务：因为可能有小数据量的批处理或存在存储过程/脚本

## 技术目标

- 利用Spring编程模式：使开发者专注于业务逻辑，让框架解决基础功能
- 在基础架构、批处理执行环境、批处理应用之间有明确的划分
- 以接口形式提供通用的核心服务，以便所有项目都能使用
- 提供简单的默认实现，以实现核心执行接口的“开箱即用”
- 易于配置、定制和扩展服务,基于spring框架的各个层面
- 所有的核心服务都可以很容易地扩展与替换，却不会影响基础系统层。
- 提供一个简单的部署模型，通过Maven编译,将应用程序与框架的JAR包完全分离

# Spring Batch架构

Spring Batch 设计时充分考虑了可扩展性和各类终端用户。下图显示了Spring Batch的架构层次示意图,这种架构层次为终端用户开发者提供了很好的扩展性与易用性.

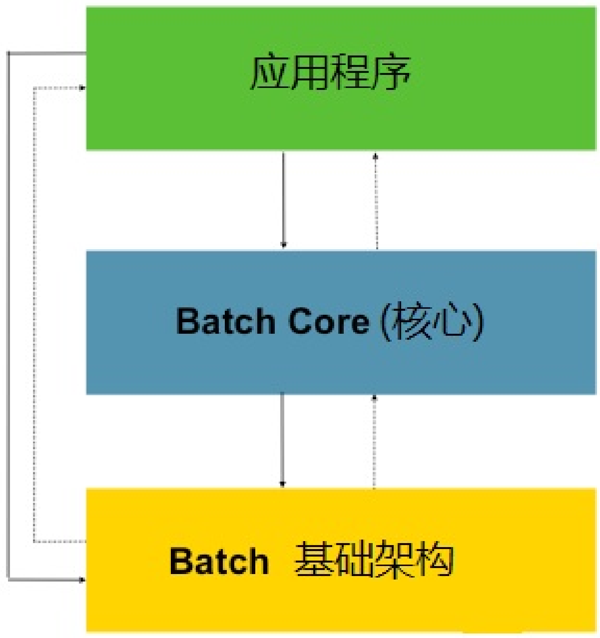



图1.1: Spring Batch 分层架构

Spring Batch 架构主要分为三类高级组件: 应用层(Application), 核心层(Core) 和基础架构层(Infrastructure)。

应用层(Application)包括开发人员用Spring batch编写的所有批处理作业和自定义代码。

Batch核心(Batch Core) 包含加载和控制批处理作业所必需的核心类。包括 **JobLauncher**, **Job**, 和 **Step** 的实现.

应用层(Application) 与 核心等(Core)都构建在通用基础架构层之上. 基础架构包括通用的 readers(**ItemReader**) 和 writers( **ItemWriter**), 以及 services (如重试模块 **RetryTemplate**), 可以被应用层和核心层所使用.

# 通用批处理的指导原则

下面是一些关键的指导原则,在构建批处理解决方案可以参考:

- 批处理架构通常会影响在线服务的架构,反之亦然。设计架构和环境时请尽可能使用公共的模块。
- 尽可能的简化,避免在单个批处理应用中构建复杂的逻辑结构。
- 尽可能在数据存放的地方处理这些数据,反之亦然(即，各自负责处理自己的数据)。
- 尽可能少的使用系统资源,尤其是I/O。尽可能多地在内存中执行大部分操作。
- 审查应用程序I/O(分析SQL语句)以避免不必要的物理I/O。特别是以下四个常见的缺陷(flaws)需要避免:
    - 在每个事务中都将(所有并不需要的)数据读取,并缓存起来;
    - 多次读取/查询同一事务中已经读取过的数据;
    - 引起不必要的表或索引扫描;
    - 在SQL语句的WHERE子句中不指定过滤条件。
- 在同一个批处理不要做两次一样的事。例如,如果你需要报表的数据汇总,请在处理每一条记录时使用增量来存储,尽可能不要再去遍历一次同样的数据。
- 在批处理程序开始时就分配足够的内存,以避免运行过程中再执行耗时的内存分配。
- 总是将数据完整性假定为最坏情况。插入适当的检查和数据校验以保持数据完整性(integrity)。
- 如有可能,请为内部校验实现checksum。例如,平面文件应该有一条结尾记录,说明文件中的总记录数和关键字段的集合(aggregate)。
- 尽可能早地在模拟生产环境下使用真实的数据量，进行计划和执行压力测试。
- 在大型批处理系统中,备份会是一个很大的挑战,特别是 7x24小时不间断的在线服务系统。数据库备份通常在设计时就考虑好了,但是文件备份也应该提升到同养的重要程度。如果系统依赖于文本文件,文件备份程序不仅要正确设置和形成文档,还要定期进行测试。

# 批处理策略

为了辅助批处理系统的设计和实现、应该通过结构示意图和代码实例的形式为设计师和程序员提供基础的批处理程序构建模块和以及处理模式. 在设计批处理Job时,应该将业务逻辑分解成一系列的步骤,使每个步骤都可以利用以下的标准构建模块来实现:

- 转换程序(Conversion Applications): 由外部系统提供或需要写入到外部系统的各种类型的文件,我们都需要为其创建一个转换程序, 用来将所提供的事务记录转换成符合要求的标准格式.这种类型的批处理程序可以部分或全部由转换工具模块组成(translation utility modules)(参见 Basic Batch Services,基本批处理服务).
- 验证程序(Validation Applications): 验证程序确保所有输入/输出记录都是正确和一致的.验证通常基于文件头和结尾信息,校验和(checksums)以及记录级别的交叉验证算法.
- 提取程序(Extract Applications): 这种程序从数据库或输入文件读取一堆记录,根据预定义的规则选取记录,并将选取的记录写入到输出文件.
- 提取/更新程序(Extract/Update Applications): 这种程序从数据库或输入文件读取记录,并将输入的每条记录都更新到数据库,或记录到输出文件.
- 处理和更新程序(Processing and Updating Applications): 这种程序对从 提取或验证程序 传过来的输入事务记录进行处理.这些处理通常包括 从数据库读取数据,有可能更新数据库,并创建输出记录.
- 输出/格式化程序(Output/Format Applications): 这种程序从输入文件中读取信息,将数据重组成为标准格式,并打印到输出文件,或者传输给另一个程序或系统.

因为业务逻辑不能用上面介绍的这些标准模块来完成, 所以还需要另外提供一个基本的程序外壳(application shell).

除了这些主要的模块,每个应用还可以使用一到多个标准的实用程序环节(standard utility steps),如:

- Sort 排序,排序程序从输入文件读取记录,并根据记录中的某个key字段重新排序,然后生成输出文件. 排序通常由标准的系统实用程序来执行.
- Split 拆分,拆分程序从单个输入文件中读取记录,根据某个字段的值,将记录写入到不同的输出文件中. 拆分可以自定义或者由参数驱动的(parameter-driven)系统实用程序来执行.
- Merge 合并,合并程序从多个输入文件读取记录,并将组合后的数据写入到单个输出文件中. 合并可以自定义或者由参数驱动的(parameter-driven)系统实用程序来执行.

批处理程序也可以根据输入来源分类:

- 数据库驱动(Database-driven)的应用程序, 由从数据库中获取的行或值驱动.
- 文件驱动(File-driven)的应用程序,是由从文件中获取的值或记录驱动的.
- 消息驱动(Message-driven)的应用程序由从消息队列中检索到的消息驱动.

所有批处理系统的基础都是处理策略.影响策略选择的因素包括: 预估的批处理系统容量, 在线并发或与另一个批处理系统的并发量, 可用的批处理时间窗口(随着越来越多的企业想要全天候(7x24小时)运转,所以基本上没有明确的批处理窗口).

典型的批处理选项包括:

- 在一个批处理窗口中执行常规离线批处理
- 并发批处理/在线处理
- 同一时刻有许多不同的批处理(runs or jobs)在并行执行
- 分区(即同一时刻,有多个实例在处理同一个job)
- 上面这些的组合

上面列表中的顺序代表了批处理实现复杂性的排序,在同一个批处理窗口的处理最简单,而分区实现最复杂.

商业调度器可能支持上面的部分/或所有类型.

下面的部分将详细讨论这些处理选项.需要特别注意的是, 批处理所采用的提交和锁定策略将依赖于处理执行的类型,作为最佳

实践,在线锁策略应该使用相同的原则.因此,在设计批处理整体架构时不能简单地拍脑袋决定(译注:即需要详细的论证和分析).

锁策略可以只使用普通的数据库锁,也可以在架构中实现自定义的锁服务.锁服务将跟踪数据库锁定(例如在一个专用的数据库表(db-table)中存储必要的信息),然后在应用程序请求数据库操作时授予权限或拒绝.重试逻辑也可以通过这种架构实现,以避免批处理作业因为资源锁定的情况而失败.

**1.**在一个批处理窗口中的常规处理 对于运行在一个单独批处理窗口中的简单批处理,更新的数据对在线用户或其他批处理来说并没有实时性要求,也没有并发问题,在批处理运行完成后执行单次提交即可.

大多数情况下,一种更健壮的方法会更合适.要记住的一件事是,批处理系统会随着时间的流逝而增长,包括复杂度和需要处理的数据量.如果没有合适的锁定策略,系统仍然依赖于一个单一的提交点,则修改批处理程序会是一件痛苦的事情.因此,即使是最简单的批处理系统,也应该为重启-恢复(restart-recovery)选项考虑提交逻辑,更不用说下面涉及到的那些更复杂情况下的信息.

**2.**并发批处理**/**在线处理 批处理程序处理的数据如果会同时被在线用户更新,就不应该锁定在线用户需要的所有任何数据(不管是数据库还是文件),即使只需要锁定几秒钟的时间.还应该每处理一批事务就提交一次数据库.这减少了其他程序不可用的数据,也压缩了数据不可用的时间.

减少物理锁的另一个选择是实现一个行级的逻辑锁,通过使用乐观锁模式或悲观锁模式.

- 乐观锁假设记录争用的可能性很低.这通常意味着并发批处理和在线处理所使用的每个数据表中都有一个时间戳列.当程序读取一行进行处理时,同时也获得对应的时间戳.当程序处理完该行以后尝试更新时,在update操作的WHERE子句中使用原来的时间戳作为条件.如果时间戳相匹配,则数据和时间戳都更新成功.如果时间戳不匹配,这表明在本程序上次获取和此次更新这段时间内已经有另一个程序修改了同一条记录,因此更新不会被执行.

- 悲观锁定策略假设记录争用的可能性很高,因此在检索时需要获得一个物理锁或逻辑锁.有一种悲观逻辑锁在数据表中使用一个专用的lock-column列.当程序想要为更新目的而获取一行时,它在lock column上设置一个标志.如果为某一行设置了标志位,其他程序在试图获取同一行时将会逻辑上获取失败.当设置标志的程序更新该行时,它也同时清除标志位,允许其他程序获取该行.请注意,在初步获取和初次设置标志位这段时间内必须维护数据的完整性,比如使用数据库锁(eg., SELECT FOR UPDATE).还请注意,这种方法和物理锁都有相同的缺点,除了它在构建一个超时机制时比较容易管理,比如记录而用户去吃午餐了,则超时时间到了以后锁会被自动释放.

这些模式并不一定适用于批处理,但他们可以被用在并发批处理和在线处理的情况下(例如,数据库不支持行级锁).作为一般规则,乐观锁更适合于在线应用,而悲观锁更适合于批处理应用.只要使用了逻辑锁,那么所有访问逻辑锁保护的数据的程序都必须采用同样的方案.

请注意,这两种解决方案都只锁定(address locking)单条记录.但很多情况下我们需要锁定一组相关的记录.如果使用物理锁,你必须非常小心地管理这些以避免潜在的死锁.如果使用逻辑锁,通常最好的解决办法是创建一个逻辑锁管理器,使管理器能理解你想要保护的逻辑记录分组(groups),并确保连贯和没有死锁(non-deadlocking).这种逻辑锁管理器通常使用其私有的表来进行锁管理、争用报告、超时机制 等等.

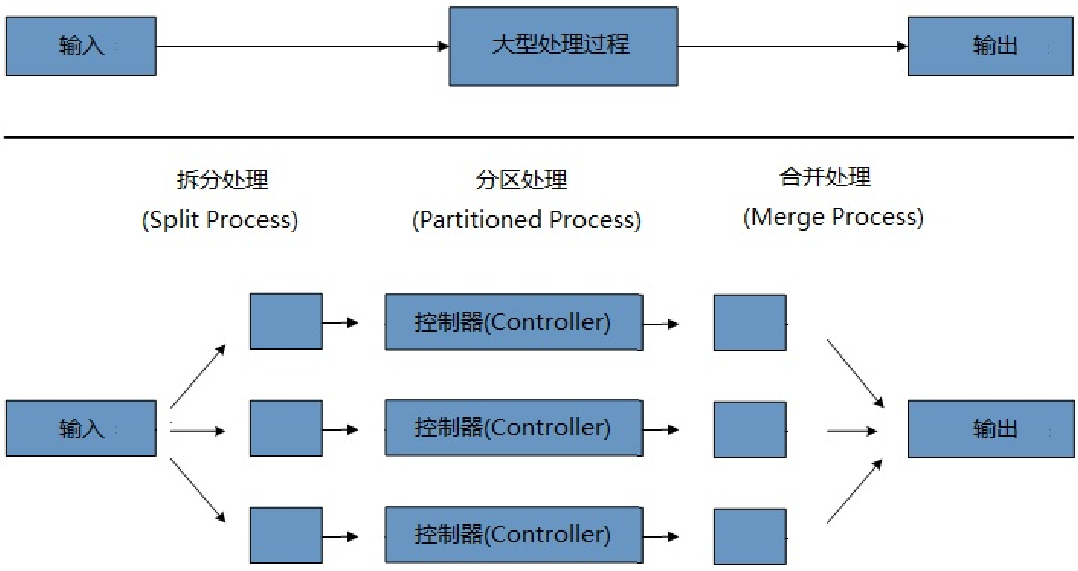
**3.**并行处理 并行处理允许多个批处理运行(run,名词,大意为运行中的程序)/任务(job)同时并行地运行,以使批处理总运行时间降到最低.如果多个任务不使用同一个文件、数据表、索引空间时这并不算什么问题.如果确实存在共享和竞争,那么这个服务就应该使用分区数据来实现.另一种选择是使用控制表来构建一个架构模块以维护他们之间的相互依赖关系.控制表应该为每个共享资源分配一行记录,不管这些资源是否被某个程序所使用.执行并行作业的批处理架构或程序随后将查询这个控制表,以确定是否可以访问所需的资源.

如果解决了数据访问的问题,并行处理就可以通过使用额外的线程来并行实现.在传统的大型主机环境中,并行作业类上通常被用来确保所有进程都有充足的CPU时间.无论如何,解决方案必须足够强劲,以确保所有正在运行的进程都有足够的时间片.

并行处理的其他关键问题还包括负载平衡以及一般系统资源的可用性(如文件、数据库缓冲池等).还要注意控制表自身也可能很容易变成一个至关重要的资源(即可能发生严重竞争?).

**4.**分区**(Partitioning)** 分区技术允许多版本的大型批处理程序并发地(concurrently)运行. 这样做的目的是减少超长批处理作业过程所需的时间. 可以成功分区的过程主要是那些可以拆分的输入文件 和/或 主要的数据库表被分区以允许程序使用不同的数据来运行.

此外,被分区的过程必须设计为只处理分配给他的数据集. 分区架构与数据库设计和数据库分区策略是密切相关的. 请注意,数据库分区并不一定指数据库需要在物理上实现分区,尽管在大多数情况下这是明智的.下面的图片展示了分区的方法:




系统架构应该足够灵活,以允许动态配置分区的数量. 自动控制和用户配置都应该纳入考虑范围. 自动配置可以根据参数来决定,例如输入文件大小 和/或 输入记录的数量.

**4.1**分区方法 下面列出了一些可能的分区方法. 选择哪种分区方法要根据具体情况来决定.

*1.*使用固定值来分解记录集

这涉及到将输入的记录集合分解成偶数个部分(例如10份,这样每部分是整个数据集的十分之一). 每个部分稍后由一个批处理/提取程序实例来处理.

为了使用这种方法,需要在预处理时将记录集拆分. 拆分的结果有一个最大值和最小值位置, 这两个值可以用作限制每个 批处理/提取程序处理部分的输入.

预处理可能是一个很大的开销,因为它必须计算并确定的每部分数据集的边界.

*2.*根据关键列*(Key Column)*分解

这涉及到将输入记录按照某个关键列来分解,比如定位码(location code),并将每个键分配给一个批处理实例.为了达到这个目标,也可以使用列值.

*3.*根据分区表决定分配给哪一个批处理实例*(*详情见下文*).*

*4.*根据值的一部分决定分配给哪个批处理实例的值*(*例如值 *0000-0999*、*1000-1999* 等*)*

在使用第1种方法时, 新值的添加将意味着需要手动重新配置批处理/提取程序,以确保新值被添加到某个特定的实例.

在使用第2种方法时,将确保所有的值都会被某个批处理作业实例处理到. 然而,一个实例处理的值的数量依赖于列值的分布(即可能存在大量的值分布在0000-0999范围内,而在1000-1999范围内的值却很少).如果使用这种方法,设计时应该考虑到数据范围的切分.

在这两种方法中,并不能将指定给批处理实例的记录实现最佳均匀分布. 批处理实例的数量并不能动态配置.

*5.*根据视图来分解

这种方法基本上是根据键列来分解,但不同的是在数据库级进行分解.它涉及到将记录集分解成视图.这些视图将被批处理程序的各个实例在处理时使用. 分解将通过数据分组来完成.

使用这个方法时,批处理的每个实例都必须为其配置一个特定的视图(而非主表).当然,对于新添加的数据,这个新的数据分组必须被包含在某个视图中.也没有自动配置功能,实例数量的变化将导致视图需要进行相应的改变.

*6.*附加的处理指示器

这涉及到输入表一个附加的新列,它充当一个指示器. 在预处理阶段,所有指示器都被标志为未处理. 在批处理程序获取记录阶段,只会读取被标记为未处理的记录,一旦他们被读取(并加锁),它们就被标记为正在处理状态. 当记录处理完成,指示器将被更新为完成或错误.批处理程序的多个实例不需要改变就可以开始,因为附加列确保每条纪录只被处理一次.

使用该选项时,表上的I/O会动态地增长.在批量更新的程序中,这种影响被降低了,因为写操作是必定要进行的.

*7.*将表提取到平面文件

这包括将表中的数据提取到一个文件中. 然后可以将这个文件拆分成多个部分,作为批处理实例的输入.

使用这个选项时,将数据提取到文件中,并将文件拆分的额外开销,有可能抵消多分区处理(multi-partitioning)的效果.可以通过改变文件分割脚本来实现动态配置.

*8.*使用哈希列*(Hashing Column)*

这个计划需要在数据库表中增加一个哈希列(key/index)来检索驱动(driver)记录. 这个哈希列将有一个指示器来确定将由批处理程序的哪个实例处理某个特定的行.例如,如果启动了三个批处理实例,那么“A”指示器将标记某行由实例1来处理,“B”将标记着将由实例2来处理,以此类推.

稍后用于检索记录的过程(procedure,程序)将有一个额外的WHERE子句来选择以一个特定指标标记的所有行. 这个表的insert需要附加的标记字段,默认值将是其中的某一个实例(例如“A”).

一个简单的批处理程序将被用来更新不同实例之间的重新分配负载的指标.当添加足够多的新行时,这个批处理会被运行(在任何时间,除了在批处理窗口中)以将新行分配给其他实例.

批处理应用程序的其他实例只需要像上面这样的批处理程序运行着以重新分配指标,以决定新实例的数量.

**4.2**数据库和应用程序设计原则

如果一个支持多分区(multi-partitioned)的应用程序架构,基于数据库采用关键列(key column)分区方法拆成的多个表,则应该包含一个中心分区仓库来存储分区参数.这种方式提供了灵活性,并保证了可维护性.这个中心仓库通常只由单个表组成,叫做分区表.

存储在分区表中的信息应该是是静态的,并且只能由DBA维护.每个多分区程序对应的单个分区有一行记录,组成这个表.这个表应该包含这些列: 程序ID编号,分区编号(分区的逻辑ID),一个分区对应的关键列（keycolumn）的最小值,分区对应的关键列的最大值.

在程序启动时,应用程序架构(Control Processing Tasklet,控制处理微线程)应该将程序id和分区号传递给该程序.这些变量被用于读取分区表,来确定应用程序应该处理的数据范围(如果使用关键列的话).另外分区号必须在整个处理过程中用来:

- 为了使合并程序正常工作,需要将分区号添加到输出文件/数据库更新
- 向框架的错误处理程序报告正常处理批处理日志和执行期间发生的所有错误

**4.3** 尽可能杜绝死锁

当程序并行或分区运行时,会导致数据库资源的争用,还可能会发生死锁(Deadlocks).其中的关键是数据库设计团队在进行数据

库设计时必须考虑尽可能消除潜在的竞争情况.

还要确保设计数据库表的索引时考虑到性能以及死锁预防.

死锁或热点往往发生在管理或架构表上,如日志表、控制表, 锁表(lock tables).这些影响也应该纳入考虑.为了确定架构可能的瓶颈,一个真实的压力测试是至关重要的.

要最小化数据冲突的影响,架构应该提供一些服务,如附加到数据库或遇到死锁时的 等待-重试(wait-and-retry)间隔时间.这意味着要有一个内置的机制来处理数据库返回码,而不是立即引发错误处理,需要等待一个预定的时间并重试执行数据库操作.

**4.4**参数传递和校验

对程序开发人员来说,分区架构应该相对透明.框架以分区模式运行时应该执行的相关任务包括:

- 在程序启动之前获取分区参数
- 在程序启动之前验证分区参数
- 在启动时将参数传递给应用程序

验证(validation)要包含必要的检查,以确保:

- 应用程序已经足够涵盖整个数据的分区
- 在各个分区之间没有遗漏断代(gaps)

如果数据库是分区的,可能需要一些额外的验证来保证单个分区不会跨越数据库的片区.

体系架构应该考虑整合分区(partitions).包括以下关键问题:

- 在进入下一个任务步骤之前是否所有的分区都必须完成?
- 如果一个分区Job中止了要怎么处理?

# Spring Batch使用示例: 读取CSV文件并写入MySQL数据库

原文链接: Reading and writing CVS files with Spring Batch and MySQL

原文作者: Steven Haines - 技术架构师

编写批处理程序来处理GB级别数据量无疑是种海啸般难以面对的任务,但我们可以用Spring Batch将其拆解为小块小块的(chunk)。 Spring Batch 是Spring框架的一个模块,专门设计来对各种类型的文件进行批量处理。 本文先讲解一个简单的作业 —— 将产品列表从CSV文件中读取出来,然后导入MySQL数据库中; 然后我们一起研究 Spring Batch 模块的批处理功能(/性能),如单/多处理单元(processors), 同时辅以多个微线程(tasklets); 最后简要介绍Spring Batch对跳过记录(skipping), 重试记录(retrying),以及批处理作业的重启(restarting )等弹性工具。

如果你曾在Java企业系统中用批处理来处理过成千上万的数据交换,那你就知道工作负载是怎么回事。 批处理系统要处理庞大无比的数据量,处理单条记录失败的情况,还要管理中断,在重启动后不要再去处理那些已经执行过的部分。

对于没有相关经验的初学者,下面是需要批处理的一些场景,并且如果使用Spring Batch 很可能会节省你很多宝贵的时间:

- 接收的文件缺少了一部分需要的信息,你需要读取并解析整个文件,调用某个服务来获得缺少的那部分信息,然后写入到某个输出文件,供其他批处理程序使用。
- 如果执行环境中发生了一个错误,则将失败信息写入数据库。 有专门的程序每隔15分钟来遍历一次失败信息,如果标记为可以重试,那就再执行一次。
- 在工作流中,你希望其他系统在收到事件消息时,来调用某个特定服务。 如果其他系统没有调用这个服务,那么一段时间后需要自动清理过期数据,以避免影响到正常的业务流程。
- 每天收到员工信息更新的文件,你需要为新员工建立相关档案和账号(artifacts)。
- 有些定制订单的服务。 你需要在每天晚上执行批处理程序来生成清单文件,并将它们发送到相应的供应商手上。

## 作业与分块: Spring Batch 范例

Spring Batch 有很多组成部分,我们先来看批量作业中的核心处理。 可以将一个作业分成下面3个不同的步骤:

1. 读取数据
2. 对数据进行各种处理
3. 对数据进行写操作

例如,我们可以打开一个CSV格式的数据文件,对文件中的数据执行某些处理,然后将数据写入数据库。 在Spring Batch中, 您需要配置一个读取程序 **reader** 来读取文件中的数据(每次一行), 然后并将每一行数据传递给 **processor** 进行处理, 处理器将会将结果收集并分组为"块 chunks", 并把这些记录发送给 **writer** ,在这里是插入到数据库中。 可以参考图1所示的周期。




图**1 Spring Batch**批处理的基本逻辑

Spring Batch实现了常见输入源的 readers, 极大地简化了批处理过程.包括 CSV文件, XML文件、数据库、文件中的JSON记录,甚至是 JMS; 同样也实现了对应的 writers。 如有需要,创建自定义的 readers and writers 也是相当简单的。

首先,让我们一起配置一个 file reader 来读取 CSV文件,将其内容映射到一个对象中,并将生成的对象插入数据库中。

下载本教程的源代码: SpringBatch-CVS演示代码

# 读取并处理CVS文件

Spring Batch 内置的reader, **org.springframework.batch.item.file.FlatFileItemReader** 将文件解析为许多单独的行。 它需要一个纯文本文件的引用,文件开头要忽略的行数(通常是头信息), 以及一个将单行转换为一个对象的 line mapper. 行映射器需要一个分割字符串的分词器,用来将一行划分为各个组成字段, 以及一个field set mapper,根据字段值构建一个对象。 **FlatFileItemReader** 的配置如下所示:

**清单1 一个Spring Batch 配置文件**

```
<bean id="productReader" class="org.springframework.batch.item.file.FlatFileItemReader" scope="step">

    <!-- <property name="resource" value="file:./sample.csv" /> -->
    <property name="resource" value="file:#{jobParameters['inputFile']}" />

    <property name="linesToSkip" value="1" />

    <property name="lineMapper">
        <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">

            <property name="lineTokenizer">
                <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">
                    <property name="names" value="id,name,description,quantity" />
                </bean>
            </property>

            <property name="fieldSetMapper">
                <bean class="com.geekcap.javaworld.springbatchexample.simple.reader.ProductFieldSetMapper" />
            </property>
        </bean>
    </property>
</bean>
```

让我们来看看这些组件。 首先,图2显示了它们之间的关系:

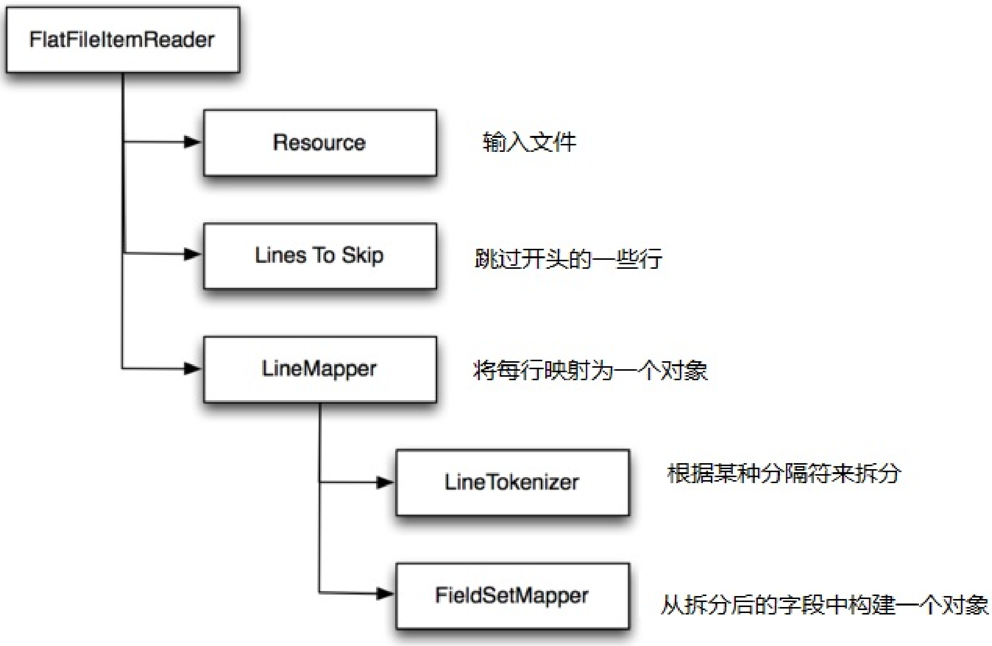



图**2 FlatFileItemReader**的组件

**Resources:** *resource* 属性指定了要读取的文件。 注释掉的 resource 使用了文件的相对路径,也就是批处理作业工作目录下的 *sample.csv* 。 作业参数 *InputFile* 就更可爱了: job parameters允许在运行时动态指定相关参数。 在使用 import 文件的情况下,在运行时才决定使用哪个参数比起在编译时就固定要灵活好用得多。 (如果要一遍又一遍,五六七八遍导入同一个文件时又会相当的无聊了!)

**Lines to skip:** *linesToSkip* 属性告诉 file reader 有多少标题行需要跳过。 CSV文件经常包含标题信息,如列名称,在文件的第一行,所以在本例中,我们让reader 跳过文件的第一行。

**Line mapper:** *lineMapper* 负责将每行记录转换成一个对象。 需要依赖两个组件:

- *LineTokenizer* 指定了如何将一行拆分为多个字段。 本例中我们列出了CSV文件中的列名。
- *fieldSetMapper* 从字段值构造一个对象。 在我们的例子中构建了一个 Product对象,属性包括 id, name, description, 以及 quantity 字段。

请注意,虽然Spring Batch为我们提供的基础框架,但我们仍需要设置字段映射的逻辑。 清单2显示了 *Product* 对象的源码,也就是我们准备构建的对象。

清单**2 Product.java**

```
package com.geekcap.javaworld.springbatchexample.simple.model;

/**
 * 代表产品的简单值对象(POJO)
 */
public class Product
{
    private int id;
    private String name;
    private String description;
    private int quantity;

    public Product() {
    }
```

```
    public Product(int id, String name, String description, int quantity) {
    this.id = id;
    this.name = name;
    this.description = description;
    this.quantity = quantity;
    }

    public int getId() {
    return id;
    }

    public void setId(int id) {
    this.id = id;
    }

    public String getName() {
    return name;
    }

    public void setName(String name) {
    this.name = name;
    }

    public String getDescription() {
    return description;
    }

    public void setDescription(String description) {
    this.description = description;
    }

    public int getQuantity() {
    return quantity;
    }

    public void setQuantity(int quantity) {
    this.quantity = quantity;
    }
}
```

Product 类是一个简单的POJO,包含4个字段。 清单3显示了 *ProductFieldSetMapper* 类的源代码。

**清单3 ProductFieldSetMapper.java**

```
package com.geekcap.javaworld.springbatchexample.simple.reader;

import com.geekcap.javaworld.springbatchexample.simple.model.Product;
import org.springframework.batch.item.file.mapping.FieldSetMapper;
import org.springframework.batch.item.file.transform.FieldSet;
import org.springframework.validation.BindException;

/**
 * 根据 CSV 文件中的字段集合构建  Product 对象
 */
public class ProductFieldSetMapper implements FieldSetMapper<Product>
{
    @Override
    public Product mapFieldSet(FieldSet fieldSet) throws BindException {
        Product product = new Product();
        product.setId( fieldSet.readInt( "id" ) );
        product.setName( fieldSet.readString( "name" ) );
        product.setDescription( fieldSet.readString( "description" ) );
        product.setQuantity( fieldSet.readInt( "quantity" ) );
        return product;
    }
}
```

*ProductFieldSetMapper* 类继承自 *fieldSetMapper* ,它只定义了一个方法: *mapFieldSet()*. mapper映射器将每一行解析成一个 *FieldSet*（包含命名好的字段), 然后传递给 mapFieldSet() 方法。 该方法负责组建一个对象来表示 CSV文件中的一行。 在本例中,我们通过 *FieldSet* 的各种 *read* 方法 构建了一个Product实例.

# 写入数据库

在读取文件之后,我们得到了一组 `Product` ,下一步就是将其写入数据库。 技术上允许我们将这些数据连接到一个 processing step,对数据做一些处理之类的,为简单起见,我们只将数据写到数据库中。 清单4显示了 **ProductItemWriter** 类的源代码。

清单4 `ProductItemWriter.java`

```
package com.geekcap.javaworld.springbatchexample.simple.writer;

import com.geekcap.javaworld.springbatchexample.simple.model.Product;
import org.springframework.batch.item.ItemWriter;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

/**
 * Writes products to a database
 */
public class ProductItemWriter implements ItemWriter<Product>
{
    private static final String GET_PRODUCT = "select * from PRODUCT where id = ?";
    private static final String INSERT_PRODUCT = "insert into PRODUCT (id,name,description,quantity) values (?,?,?,?)";
    private static final String UPDATE_PRODUCT = "update PRODUCT set name = ?, description = ?,quantity = ? where id = 

    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public void write(List<? extends Product> products) throws Exception
    {
        for( Product product : products )
        {
            List<Product> productList = jdbcTemplate.query(GET_PRODUCT, new Object[] {product.getId()}, new RowMapper<P
                @Override
                public Product mapRow( ResultSet resultSet, int rowNum ) throws SQLException {
                    Product p = new Product();
                    p.setId( resultSet.getInt( 1 ) );
                    p.setName( resultSet.getString( 2 ) );
                    p.setDescription( resultSet.getString( 3 ) );
                    p.setQuantity( resultSet.getInt( 4 ) );
                    return p;
                }
            });

            if( productList.size() > 0 )
            {
                jdbcTemplate.update( UPDATE_PRODUCT, product.getName(), product.getDescription(), product.getQuantity()
            }
            else
            {
                jdbcTemplate.update( INSERT_PRODUCT, product.getId(), product.getName(), product.getDescription(), proc
            }
        }
    }
}
```

**ProductItemWriter** 类继承(extends, 其实继承和实现 implements 没有本质区别.) **ItemWriter** 并实现了其唯一的方法: `write()` . `write()` 方法接受一个泛型继承 `Product` 的 `list` . Spring Batch 使用“chunking”策略实现其 writers , 意思就是读取时是一次执行一个item, 而写入时是将一组数据一块写。 如下面的job配置所示,您可以(通过 `commit-interval` )完全控制每次想要一起写的item的数量。 在上面的例子中, `write()` 方法做了这些事:

1. 它执行一个 **SQL** `SELECT` 语句来根据指定的 `id` 检索 **Product**.
2. 如果 `SELECT` 返回一条记录, 则 write() 中执行一个 `update` 使用新value来更新数据库中的记录.

3. 如果 `SELECT` 没有返回记录, 则 write() 执行 `INSERT` 将产品信息添加到数据库中.

**ProductItemWriter** 类使用Spring的 `JdbcTemplate` 类,它在 `applicationContext.xml` 文件中定义并通过自动装配机制注入到 **ProductItemWriter** 类。 如果你没有用过 Jdbctemplate 类,可以把它理解为是 JDBC 接口的一个封装. 与数据库进行交互的 模板设计模式 的实现. 代码应该很容易读懂, 如果你想了解更多信息, 请查看 SpringJdbcTemplate 的 javadoc。

# 与 application context 文件组装

到目前为止我们已经建立了一个 `Product` 领域对象, 一个 `ProductFieldSetMapper` 类, 用来将CSV文件中的每一行转换为一个对象, 以及一个 `ProductItemWriter` 类, 来将对象写入数据库。 下面我们需要配置 Spring Batch 来将这些东西组装在一起。 清单5 显示了 `applicationContext.xml` 文件的源代码, 这里面定义了我们需要的bean。

清单 **5.** **`applicationContext.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xmlns:jdbc="http://www.springframework.org/schema/jdbc"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spri
                http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-cont
                http://www.springframework.org/schema/batch http://www.springframework.org/schema/batch/spring-batch.xs
                http://www.springframework.org/schema/jdbc http://www.springframework.org/schema/jdbc/spring-jdbc.xsd">


    <context:annotation-config />

    <!-- Component scan to find all Spring components -->
    <context:component-scan base-package="com.geekcap.javaworld.springbatchexample" />


    <!-- Data source - connect to a MySQL instance running on the local machine -->
    <bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource" destroy-method="close">
        <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
        <property name="url" value="jdbc:mysql://localhost/spring_batch_example"/>
        <property name="username" value="sbe"/>
        <property name="password" value="sbe"/>
    </bean>

    <bean id="transactionManager" class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="jdbcTemplate" class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <!-- Create job-meta tables automatically -->
    <jdbc:initialize-database data-source="dataSource">
        <jdbc:script location="org/springframework/batch/core/schema-drop-mysql.sql" />
        <jdbc:script location="org/springframework/batch/core/schema-mysql.sql" />
    </jdbc:initialize-database>


    <!-- Job Repository: used to persist the state of the batch job -->
    <bean id="jobRepository" class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
        <property name="transactionManager" ref="transactionManager" />
    </bean>


    <!-- Job Launcher: creates the job and the job state before launching it -->
    <bean id="jobLauncher" class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
        <property name="jobRepository" ref="jobRepository" />
    </bean>


    <!-- Reader bean for our simple CSV example -->
    <bean id="productReader" class="org.springframework.batch.item.file.FlatFileItemReader" scope="step">
```

```
        <!-- <property name="resource" value="file:./sample.csv" /> -->
        <property name="resource" value="file:#{jobParameters['inputFile']}" />


        <!-- Skip the first line of the file because this is the header that defines the fields -->
        <property name="linesToSkip" value="1" />

        <!-- Defines how we map lines to objects -->
        <property name="lineMapper">
            <bean class="org.springframework.batch.item.file.mapping.DefaultLineMapper">

                <!-- The lineTokenizer divides individual lines up into units of work -->
                <property name="lineTokenizer">
                    <bean class="org.springframework.batch.item.file.transform.DelimitedLineTokenizer">

                        <!-- Names of the CSV columns -->
                        <property name="names" value="id,name,description,quantity" />
                    </bean>
                </property>

                <!-- The fieldSetMapper maps a line in the file to a Product object -->
                <property name="fieldSetMapper">
                    <bean class="com.geekcap.javaworld.springbatchexample.simple.reader.ProductFieldSetMapper" />
                </property>
            </bean>
        </property>
    </bean>

    <bean id="productWriter" class="com.geekcap.javaworld.springbatchexample.simple.writer.ProductItemWriter" />

</beans>
```

注意,将 job 配置从 application/environment 中分离出来使我们能够将 job 从一个环境移到另一个环境 而不需要重新定义一个job。 清单5中定义了下面这些bean:

- **dataSource** : 示例程序连接到MySQL,所以数据库连接池配置为连接到一个名为 `spring_batch_example` 的MySQL数据库， 地址为本机(localhost),具体设置参见下文。
- **transactionmanager** : Spring事务管理器, 用于管理MySQL事务。
- **jdbctemplate** : 该类提供了与JDBC connections交互的模板设计模式实现。 这是一个 Helper 类,用来简化我们使用数据库。 在实际的项目中一般会使用某种ORM工具, 例如Hibernate,上面再包装一个服务层, 但本示例中我想让它尽可能地简单。
- **jobrepository** : `MapJobRepositoryFactoryBean` 是 Spring Batch 管理 job 状态的组件。 在这里它使用前面配置的 jdbctemplate 将 job 信息存储到MySQL数据库中。
- **jobLauncher** : 这是启动和管理 Spring Batch 作业工作流的组件,。
- **productReader** : 在job中这个 bean 负责执行读操作。
- **productWriter** : 这个bean 负责将 `Product` 实例写入数据库。

请注意, `jdbc:initialize-database` 节点包含了两个用来创建所需数据库表的Spring Batch 脚本。 这些脚本我文件位于 Spring Batch core 的JAR文件中(由Maven自动引入了)对应的路径下。 JAR文件中包含了许多数据库对应的脚本, 比如 MySQL、Oracle、SQL Server,等等。 这些脚本负责在运行 job 时创建需要的schema。 在本示例中,它删除(drop)表,然后再创建(create)表, 你可以试着运行一下。 如果在生产环境中, 你应该将SQL文件提取出来,然后手动执行 —— 毕竟生产环境一般创建了就不会删除。

**Spring Batch 中的 Lazy scope**

你可能已经注意到 `productReader` 这个bean 指定了为一个值为“`step`”的 `scope` 属性。 `step scope` 是Spring框架的 作用域

之一, 主要用于 Spring Batch。 它本质上是一个 *lazy scope*, 告诉Spring在首次访问时才创建bean。 在本例中, 我们需要使用 step scope 是因为使用了 job 参数的 "`InputFile`" 值, 这个值在应用程序启动时是不存在的。 使用 step scope 使Spring Batch在创建这个 bean 时能够找到 "`InputFile`" 值。

## 定义job

清单6显示了 `file-import-job.xml` 文件, 该文件定义了实际的 job 作业。

**清单6 `file-import-job.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spri
                http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-cont
                http://www.springframework.org/schema/batch http://www.springframework.org/schema/batch/spring-batch.xs

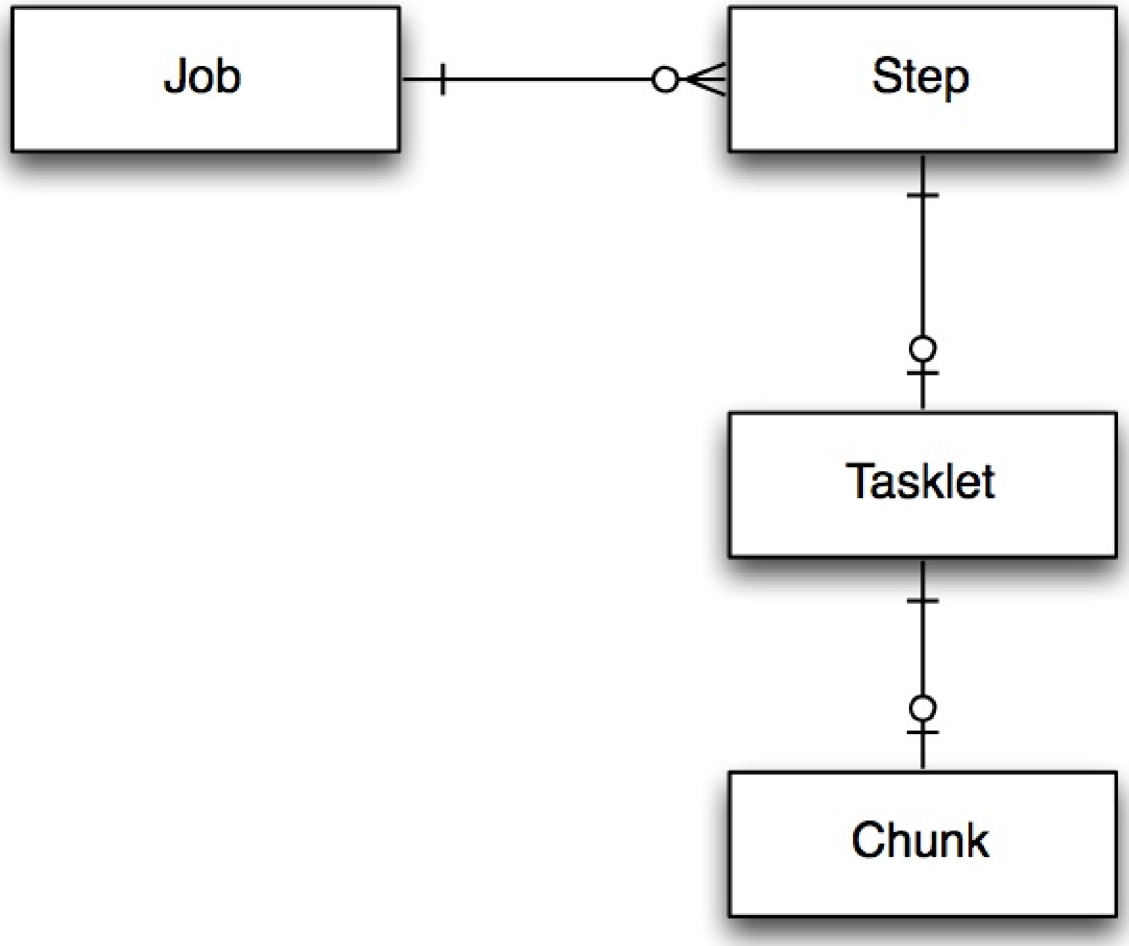

    <!-- Import our beans -->
    <import resource="classpath:/applicationContext.xml" />

    <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="importFileStep">
            <tasklet>
                <chunk reader="productReader" writer="productWriter" commit-interval="5" />
            </tasklet>
        </step>
    </job>

</beans>
```

请注意, 一个 job 可以包含 0 到 多个 step; 一个 step 可以包含 0 到 多个 tasklet; 一个 tasklet 可以包含 0 到多个 chunk, 如图3所示。

**图3 Jobs, steps, tasklets 和 chunks的关系**




在我们的示例中, **simpleFileImportJob** 包含一个名为 `importFileStep` 的step。 **importFileStep** 包含一个未命名的 tasklet, tasklet又包含有一个 chunk。 chunk 引用了 `productReader` 和 `productWriter` 。 同时指定了一个属性 **commit-interval**, 值为 `5` . 意思是每5条记录就调用一次 writer。 该 step 利用 `productReader` 一次读取5条产品记录, 然后将这些记录传递给 `productWriter` 写出。 这一块一直重复执行, 直到所有数据都处理完成为止。

清单6 还还引入了 `applicationContext.xml` 文件,该文件包含所有需要的bean。 而 Jobs 通常在单独的文件中定义; 这是因为 job 加载器在执行时需要一个 job 文件以及对应的 job name。 虽然可以讲所有的东西揉进一个文件中,但很快变得臃肿难以维护,所以一般约定, 一个 job 定义在一个文件中, 同时引入所有依赖文件。

最后,你可能会注意到,job 节点 上定义了XML名称空间( `xmlns` ) 。 这样做是为了不想在每个节点上再加上前缀 " `batch:` "。 在节点级别定义的 namespace 会在该节点和所有子节点上生效。

## 构建并运行项目

清单7显示了构建此示例项目的POM文件的内容

**清单7 `pom.xml`**

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.geekcap.javaworld</groupId>
  <artifactId>spring-batch-example</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
```

```
    <name>spring-batch-example</name>
    <url>http://maven.apache.org</url>

  <properties>
      <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
      <spring.version>3.2.1.RELEASE</spring.version>
      <spring.batch.version>2.2.1.RELEASE</spring.batch.version>
      <java.version>1.6</java.version>
  </properties>

  <dependencies>
      <!-- Spring Dependencies -->
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-context</artifactId>
          <version>${spring.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-core</artifactId>
          <version>${spring.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-beans</artifactId>
          <version>${spring.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework</groupId>
          <artifactId>spring-jdbc</artifactId>
          <version>${spring.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework.batch</groupId>
          <artifactId>spring-batch-core</artifactId>
          <version>${spring.batch.version}</version>
      </dependency>
      <dependency>
          <groupId>org.springframework.batch</groupId>
          <artifactId>spring-batch-infrastructure</artifactId>
          <version>${spring.batch.version}</version>
      </dependency>

      <!-- Apache DBCP-->
      <dependency>
          <groupId>commons-dbcp</groupId>
          <artifactId>commons-dbcp</artifactId>
          <version>1.4</version>
      </dependency>

      <!-- MySQL -->
      <dependency>
          <groupId>mysql</groupId>
          <artifactId>mysql-connector-java</artifactId>
          <version>5.1.27</version>
      </dependency>


      <!-- Testing -->
      <dependency>
          <groupId>junit</groupId>
          <artifactId>junit</artifactId>
          <version>4.11</version>
          <scope>test</scope>
      </dependency>
  </dependencies>

  <build>
      <plugins>
          <plugin>
              <groupId>org.apache.maven.plugins</groupId>
              <artifactId>maven-compiler-plugin</artifactId>
              <configuration>
                  <source>${java.version}</source>
                  <target>${java.version}</target>
              </configuration>
          </plugin>
          <plugin>
```

```
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <archive>
                        <manifest>
                            <addClasspath>true</addClasspath>
                            <classpathPrefix>lib/</classpathPrefix>
                        </manifest>
                    </archive>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-dependency-plugin</artifactId>
                <executions>
                    <execution>
                        <id>copy</id>
                        <phase>install</phase>
                        <goals>
                            <goal>copy-dependencies</goal>
                        </goals>
                        <configuration>
                            <outputDirectory>${project.build.directory}/lib</outputDirectory>
                        </configuration>
                    </execution>
                </executions>
            </plugin>
        </plugins>
        <finalName>spring-batch-example</finalName>
    </build>


</project>
```

上面的POM文件先引入了 Spring context, core, beans, 和 JDBC 框架/类库， 然后引入 Spring Batch core 以及 infrastructure 依赖(包)。 这些依赖项就是 Spring 和 Spring Batch的基础。 当然也引入了 Apache DBCP, 使我们能构建数据库连接池和MySQL驱动。 `plug-in` 部分指定了使用Java 1.6进行编译,并在 build 时将所有依赖项库复制到 lib 目录下。 我们可以使用下面的命令来构建项目:

```
mvn clean install
```

# Spring Batch连接到一个数据库

现在我们的 job 已经设置好了, 如果想在生产环境中运行还需要将Spring Batch连接到数据库。 Spring Batch 需要一些表, 用来记录 job 的当前状态和已经处理过的 record 列表。 这样,如果某个 job 确实需要重启, 则可以从上次断开的地方继续执行。

Spring Batch 可以连接到任何你喜欢的数据库, 但为了演示方便, 我们在本示例中使用MySQL。 请 下载MySQL 并安装后再执行下面的脚本。 社区版是免费的,而且能满足大多数人的需要。 请根据你的操作系统选择合适的版本下载安装. 然后可能需要手动启动MySQL(Windows 一般自动启动)。

安装好MySQL后还需要创建数据库以及相应的用户(并赋予权限)。 启动命令行并进入MySQL的bin目录启动 mysql 客户端, 连接服务器后执行以下SQL命令(请注意,在Linux下可能需要使用 `root` 用户执行 `mysql` 客户端程序, 或者使用 `sudo` 进行权限切换.

```
create database spring_batch_example;
create user 'sbe'@'localhost' identified by 'sbe';
grant all on spring_batch_example.* to 'sbe'@'localhost';
```

第一行SQL创建了一个名为 `spring_batch_example` 的数据库(database), 这个库用来保存我们的 products 信息。 第二行创建了一个名为 `sbe` 的用户 Spring Batch Example的缩写,你也可以使用其他名字,只要配置得一致就行), 密码 也指定为 `sbe` 。 最后一行将 `spring_batch_example` 数据库上的所有权限赋予 `sbe` 用户。

接下来,使用下面的命令创建 **PRODUCT** 表:

```
CREATE TABLE PRODUCT (
    ID INT NOT NULL,
    NAME VARCHAR(128) NOT NULL,
    DESCRIPTION VARCHAR(128),
    QUANTITY INT,
    PRIMARY KEY(ID)
);
```

接着,我们在项目的 target 目录下创建一个文件 `sample.csv`, 并填充一些数据(用英文逗号分隔):

```
id,name,description,quantity
1,Product One,This is product 1, 10
2,Product Two,This is product 2, 20
3,Product Three,This is product 3, 30
4,Product Four,This is product 4, 20
5,Product Five,This is product 5, 10
6,Product Six,This is product 6, 50
7,Product Seven,This is product 7, 80
8,Product Eight,This is product 8, 90
```

可以使用下面的命令启动 batch job:

```
java -cp spring-batch-example.jar:./lib/* org.springframework.batch.core.launch.support.CommandLineJobRunner classpath:
```

`CommandLineJobRunner` 是 Spring Batch 框架中执行 job 的类。 它需要定义了 job的XML文件的名称, 需要执行的job的名称, 以及其他可选的一些自定义参数。 因为 `file-import-job.xml` 在JAR文件的内部, 所以可以使用这种方式访问: `classpath:/jobs/file-import-job.xml`。 我们给需要执行的 Job指定了一个名称 `simpleFileImportJob` 并传入一个参数 `InputFile`, 值为 `sample.csv`。

如果执行不出错, 输出结果类似于下面这样:

```
Nov 12, 2013 4:09:17 PM org.springframework.context.support.AbstractApplicationContext prepareRefresh
INFO: Refreshing org.springframework.context.support.ClassPathXmlApplicationContext@6b4da8f4: startup date [Tue Nov 12
Nov 12, 2013 4:09:17 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [jobs/file-import-job.xml]
Nov 12, 2013 4:09:18 PM org.springframework.beans.factory.xml.XmlBeanDefinitionReader loadBeanDefinitions
INFO: Loading XML bean definitions from class path resource [applicationContext.xml]
Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition
INFO: Overriding bean definition for bean 'simpleFileImportJob': replacing [Generic bean: class [org.springframework.ba
Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory registerBeanDefinition
INFO: Overriding bean definition for bean 'productReader': replacing [Generic bean: class [org.springframework.batch.it
Nov 12, 2013 4:09:19 PM org.springframework.beans.factory.support.DefaultListableBeanFactory preInstantiateSingletons
INFO: Pre-instantiating singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@6aba4211: de
Nov 12, 2013 4:09:19 PM org.springframework.batch.core.launch.support.SimpleJobLauncher afterPropertiesSet
INFO: No TaskExecutor has been set, defaulting to synchronous executor.
Nov 12, 2013 4:09:22 PM org.springframework.batch.core.launch.support.SimpleJobLauncher$1 run
INFO: Job: [FlowJob: [name=simpleFileImportJob]] launched with the following parameters: [{inputFile=sample.csv}]
Nov 12, 2013 4:09:22 PM org.springframework.batch.core.job.SimpleStepHandler handleStep
INFO: Executing step: [importFileStep]
Nov 12, 2013 4:09:22 PM org.springframework.batch.core.launch.support.SimpleJobLauncher$1 run
INFO: Job: [FlowJob: [name=simpleFileImportJob]] completed with the following parameters: [{inputFile=sample.csv}] and
Nov 12, 2013 4:09:22 PM org.springframework.context.support.AbstractApplicationContext doClose
INFO: Closing org.springframework.context.support.ClassPathXmlApplicationContext@6b4da8f4: startup date [Tue Nov 12 16:
Nov 12, 2013 4:09:22 PM org.springframework.beans.factory.support.DefaultSingletonBeanRegistry destroySingletons
INFO: Destroying singletons in org.springframework.beans.factory.support.DefaultListableBeanFactory@6aba4211: defining
```

然后到数据库中检测一下 **PRODUCT** 表中是否正确保存了我们在 csv中指定的那几条记录(示例是8条)。

# 对 Spring Batch 执行批量处理

到这一步, 我们的示例程序已经从CSV文件中读取数据,并将信息导入到了数据库中。 虽然可以运行起来, 但有时候想要对数据进行转换或着过滤掉某些数据,然后再插入到数据库中。 在本节中,我们将创建一个简单的 processor , 并不覆盖原有的 product 数量,而是先从数据库中查询现有记录, 然后将CSV文件中对应的数量添加到 product 中, 然后再写入数据库。

清单8显示了 **ProductItemProcessor** 类的源代码。

清单8 `ProductItemProcessor.java`

```
package com.geekcap.javaworld.springbatchexample.simple.processor;

import com.geekcap.javaworld.springbatchexample.simple.model.Product;
import org.springframework.batch.item.ItemProcessor;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

/**
 * Processor that finds existing products and updates a product quantity appropriately
 */
public class ProductItemProcessor implements ItemProcessor<Product,Product>
{
    private static final String GET_PRODUCT = "select * from PRODUCT where id = ?";
    @Autowired
    private JdbcTemplate jdbcTemplate;

    @Override
    public Product process(Product product) throws Exception
    {
        // Retrieve the product from the database
        List<Product> productList = jdbcTemplate.query(GET_PRODUCT, new Object[] {product.getId()}, new RowMapper<Produ
            @Override
            public Product mapRow( ResultSet resultSet, int rowNum ) throws SQLException {
                Product p = new Product();
                p.setId( resultSet.getInt( 1 ) );
                p.setName( resultSet.getString( 2 ) );
                p.setDescription( resultSet.getString( 3 ) );
                p.setQuantity( resultSet.getInt( 4 ) );
                return p;
            }
        });

        if( productList.size() > 0 )
        {
            // Add the new quantity to the existing quantity
            Product existingProduct = productList.get( 0 );
            product.setQuantity( existingProduct.getQuantity() + product.getQuantity() );
        }

        // Return the (possibly) update prduct
        return product;
    }
}
```

`ProductItemProcessor` 实现的接口 `ItemProcessor<I,O>` , 其中类型 **I** 表示传递给处理器的对象类型, 而 **O** 则表示处理器返回的对象类型。 在本例中,我们传入一个 `Product` 对象,返回的也是一个 Product 对象。 **ItemProcessor** 接口只定义了一个方法: `process()` , 在里面我们根据给定的 `id` 执行一条 **SELECT** 语句从数据库中获取对应的 `Product` 。 如果找到 `Product` 对象 , 则将该对象的数量加上新的数量。

processor 没有做任何过滤,但如果 `process()` 方法返回 `null` , 则Spring Batch 将会忽略这个 item, 不将其发送给 writer.

将 processor 组装到 job 中是非常简单的。 首先,添加一个新的bean 到 `applicationContext.xml` 文件中:

```
<bean id="productProcessor" class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductItemProcessor" />
```

接下来,在 `chunk` 中通过 `processor` 属性来引用这个 bean:

```
<job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="importFileStep">
        <tasklet>
            <chunk reader="productReader" processor="productProcessor" writer="productWriter" commit-interval="5" />
        </tasklet>
    </step>
</job>
```

编译并执行 job, 如果不出错, 就可以在数据库中看到产品的数量发生了变化。

## 创建多个processors

前面我们定义了单个处理器,但某些情况下可能想要以适当的粒度来创建多个 item processor，然后按顺序在同一个 chunk之中执行. 例如,可能需要一个过滤器来跳过数据库中不存在的记录,还需要一个 processor 来正确地管理 item 数量。 这时候, 我们可以使用Spring Batch中的 `CompositeItemProcessor` 来大显身手. 使用步骤如下:

1. 创建 processor 类
2. 在applicationContext.xml中配置 bean
3. 定义一个类型为 `org.springframework.batch.item.support.CompositeItemProcessor` 的 bean,然后将其 `delegates` 设置为你想执行的处理器bean的 list
4. 让 `chunk` 的 `processor` 属性引用 `CompositeItemProcessor`

假设我们有一个 `ProductFilterProcessor` , 则可以像下面这样指定 process :

```
<bean id="productFilterProcessor" class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductFilterItemPro

<bean id="productProcessor" class="com.geekcap.javaworld.springbatchexample.simple.processor.ProductItemProcessor" />

<bean id="productCompositeProcessor" class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <ref bean="productFilterProcessor" />
            <ref bean="productProcessor" />
        </list>
    </property>
</bean>
```

然后只需修改一下 job 配置即可,如下所示:

```
<job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="importFileStep">
        <tasklet>
            <chunk reader="productReader" processor="productCompositeProcessor" writer="productWriter" commit-interval=
        </tasklet>
    </step>
</job>
```

# Tasklets(微线程)

分块是一个非常好的策略,用来将 作业拆分成多块: 依次读取每一个 item , 执行处理, 然后将其按块写出。 但如果想执行某些只需要执行一次的线性操作该怎么办呢? 此时我们可以创建一个 `tasklet` 。 `tasklet` 可以执行各种操作/需求! 例如, 可以从FTP站点下载文件, 解压/解密文件, 或者调用web服务来判断文件处理是否已经执行。 下面是创建一个 `tasklet` 的基本过程:

1. 定义一个实现 `org.springframework.batch.core.step.tasklet.Tasklet` 接口的类。
2. 实现 `execute()` 方法。
3. 返回恰当的 `org.springframework.batch.repeat.RepeatStatus` 值: `CONTINUABLE` 或者是 `FINISHED` .
4. 在 `applicationContext.xml` 文件中定义对应的 bean。
5. 创建一个 `step` , 其中有一个子元素 `tasklet` 引用第4步定义的bean。

清单9 显示了一个新的 tasklet 的源码, 将我们的输入文件拷贝到存档目录中。

清单**9** **`ArchiveProductImportFileTasklet.java`**

```
package com.geekcap.javaworld.springbatchexample.simple.tasklet;

import org.apache.commons.io.FileUtils;
import org.springframework.batch.core.StepContribution;
import org.springframework.batch.core.scope.context.ChunkContext;
import org.springframework.batch.core.step.tasklet.Tasklet;
import org.springframework.batch.repeat.RepeatStatus;

import java.io.File;

/**
 * A tasklet that archives the input file
 */
public class ArchiveProductImportFileTasklet implements Tasklet
{
    private String inputFile;

    @Override
    public RepeatStatus execute(StepContribution stepContribution, ChunkContext chunkContext) throws Exception
    {
        // Make our destination directory and copy our input file to it
        File archiveDir = new File( "archive" );
        FileUtils.forceMkdir( archiveDir );
        FileUtils.copyFileToDirectory( new File( inputFile ), archiveDir );

        // We're done...
        return RepeatStatus.FINISHED;
    }

    public String getInputFile() {
        return inputFile;
    }

    public void setInputFile(String inputFile) {
        this.inputFile = inputFile;
    }
}
```

**ArchiveProductImportFileTasklet** 类实现了 `Tasklet` 接口, 并实现了 `execute()` 方法。 其中使用Apache Commons I/O 工具库的 `FileUtils` 类来创建一个新的 `archive` 目录,然后将input file 拷贝到里面。

将下面的 bean添加到 `applicationContext.xml` 文件中:

```
<bean id="archiveFileTasklet" class="com.geekcap.javaworld.springbatchexample.simple.tasklet.ArchiveProductImportFileTa
    <property name="inputFile" value="#{jobParameters['inputFile']}" />
</bean>
```

注意, 我们传入了一个名为 `inputFile` 的 job 参数, 这个bean 设置了作用域范围 `scope="step"`, 以确保在 bean 对象创建之前需要的 job 参数都被定义。

清单10 显示了更新后的job.

清单**10** **`file-import-job.xml`**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:batch="http://www.springframework.org/schema/batch"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spri
                http://www.springframework.org/schema/context http://www.springframework.org/schema/context/spring-cont
                http://www.springframework.org/schema/batch http://www.springframework.org/schema/batch/spring-batch.xs


    <!-- Import our beans -->
    <import resource="classpath:/applicationContext.xml" />

    <job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
        <step id="importFileStep" next="archiveFileStep">
            <tasklet>
                <chunk reader="productReader" processor="productProcessor" writer="productWriter" commit-interval="5" /
            </tasklet>
        </step>
        <step id="archiveFileStep">
            <tasklet ref="archiveFileTasklet" />
        </step>
    </job>

</beans>
```

清单10中添加了一个新的step, id为 `archiveFileStep`, 然后在 `importFileStep` 中将 `"next"` 指向他。 `"next"` 参数允许我们控制job 中 step 的执行流程。 虽然超出了本文所需的范围,但我们需要注意, 可以根据某个任务的执行结果状态来决定下面执行哪个 step[也就是 job分支,类似于 if,switch 什么的]. 的 `archiveFileStep` 只包含上面创建的那个 `tasklet`。

# 弹性(Resiliency)

Spring Batch job resiliency提供了以下三个工具:

1. **Skip** : 如果处理过程中某条记录是错误的, 如CSV文件中格式不正确的行, 那么可以直接跳过该对象, 继续处理下一个。
2. **Retry** : 如果出现错误,而很可能在几毫秒后再次执行就能解决, 那么可以让 Spring Batch 对该元素重试一次/(或多次)。例如, 你可能想要更新数据库中的某条, 但另一个查询把这条记录给锁了的情况。 而根据业务设计,这个锁将会很快被释放, 而重新尝试可能就会成功。
3. **Restart** : 如果将 job 状态存储在数据库中, 而一旦它执行失败, 那么就可以选择重启 job 实例, 并继续上次的执行位置。

我们这里不会详细讲述每个 Resiliency 特征, 但我想总结一下可用的选项。

## Skipping Items(跳过某项)

有时你可能想要跳过某些记录, 比如 reader 读取的无效记录,或者处理/写入过程中出现异常的对象。 要这样做, 我们可以指定两个地方:

- 在 `chunk` 元素上定义 `skip-limit` 属性, 告诉Spring 最多允许跳过多少个 items,超过则 job 失败(如果无效记录很少那你可以接受,但如果无效记录太多,那可能输入数据就有问题了)。
- 定义一个 `skippable-exception-classes` 列表, 用来判断当前记录是否可以跳过, 可以指定 `include` 元素来决定哪些异常发生时将会跳过当前记录, 还可以指定 `exclude` 元素来决定哪些异常不会触发 skip( 比如你想跳过某个异常层次父类, 但排除一或多个子类异常时)。

例如:

```
<job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="importFileStep">
        <tasklet>
            <chunk reader="productReader" processor="productProcessor" writer="productWriter" commit-interval="5" skip-
        <skippable-exception-classes>
            <include class="org.springframework.batch.item.file.FlatFileParseException" />
        </skippable-exception-classes>
    </chunk>
        </tasklet>
    </step>
</job>
```

在这种情况下, 在处理某条记录时如果抛出 **FlatFileParseException** 异常, 则这条记录将被跳过。 如果超过10次 skip, 那么 job 失败。

## 重试（Retrying Items）

在其他情况下, 有时发生的异常是可以重试的, 如由于数据库锁导致的失败。 重试(Retry)的实现和跳过(Skip)非常相似:

- 在 `chunk` 元素上定义 `retry-limit` 属性, 告诉Spring 每个 item 最多允许重试多少次, 超过则认为该记录处理失败。 如果不将重试与跳过组合起来使用,则某条记录处理失败, 则 job也被标记为失败。
- 定义一个 `retryable-exception-classes` 列表, 用来判断当前记录是否可以重试; 可以指定 `include` 元素来决定哪些异常发生时当前记录可以重试, 还可以指定 `exclude` 元素来决定哪些异常不会重试当前记录.。

例如:

```
<job id="simpleFileImportJob" xmlns="http://www.springframework.org/schema/batch">
    <step id="importFileStep">
        <tasklet>
            <chunk reader="productReader" processor="productProcessor" writer="productWriter" commit-interval="5" retry
        <retryable-exception-classes>
            <include class="org.springframework.dao.OptimisticLockingFailureException" />
        </retryable-exception-classes>
    </chunk>
        </tasklet>
    </step>
</job>
```

还可以将重试和可跳过的异常通过对应的 skippable exception class 与 retry exception 组合起来。 因此, 如果某个异常触发了5次重试, 5次重试之后, 如果该异常也在 skippable 列表中, 那么这条记录将被跳过。 如果 exception 不在 skippable列表则会导致整个 job 失败。

## 重启 job

最后, 对于执行失败的 job作业, 我们可以重新启动,并让他们从上次断开的地方继续执行。 要达到这一点, 只需要使用和上次一模一样的参数来启动 job, 则 Spring Batch 会自动从数据库中找到这个实例然后继续执行。 你也可以拒绝重启, 或者参数控制某个 job 中的 一个 step 可以重启的次数(一般来说多次重试都失败了,那我们可能需要放弃。)

# 总结

某些业务问题使用批处理是最实在的解决方案, 而 Spring batch 框架提供了实现批处理作业的架构。 Spring Batch 将一个分块模式定义为三个阶段: 读取(read)、 处理(process)、 已经写入(write),并且支持对常见资源的读取和写入。 本期的Open source Java projects 系列探讨了 Spring Batch 是干什么的以及如何使用它。

我们先创建了一个简单的 job 从CSV文件读取 Product信息然后导入到数据库, 接着添加 processor 来对 job 进行扩展: 用来管理 product 数量。 最后我们写了一个单独的 tasklet 来归档输入文件。 虽然不是示例的一部分, 但Spring Batch 的弹性特征是非常重要的, 所以我快速介绍了Spring Batch提供的三大弹性工具: skipping records, retrying records, 和 restarting batch jobs。

本文只是简单介绍 Spring Batch 的皮毛, 但希望能让你对使用 Spring Batch 执行批处理作业有一定的了解和认识。

# Spring Batch 3.0新特性

Spring Batch 3.0 release 主要有5个主题

- JSR-252 的支持
- 支持升级至 Spring4 和 java8
- 增强 Spring Batch 之间的整合
- 支持 JobScope
- 支持 SQLite

# JSR-352支持

JSR-352是java批处理的新规范。受Spring Batch的深度影响，该规范提供了Spring Batch已经存在的相关功能。Spring Batch 3.0已实现该规范，支持遵循标准定义批处理任务了。使用JSR-352 的任务规范语言（JSL）配置一个批处理任务，代码如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="myJob3" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
    <step id="step1" >
        <batchlet ref="testBatchlet" />
    </step>
</job>
```

详细请参见 JSR-352 Support 章节

# 改进的Spring Batch Integration模块

Spring Batch Integration 曾经是 Spring Batch Admin 项目的子模块。Spring Batch的Spring Integration提供了更好整合能力的功能。这些特殊功能包括：

- 通过消息启动任务
- 异步ItemProcessors
- 提供反馈信息的消息
- 通过远程分区和远程块执行外部化批处理

详细请参见 Spring Batch Integration 章节

# 升级到支持Spring 4和Java 8

随着 Spring Batch Integration 成为 Spring Batch 项目的一个模块，它将更新为使用 Spring Integration 4。Spring Integration 4 给Spring引进了核心消息api。由此，Spring Batch 3 需要 Spring 4或更高版本的支持。

作为依赖此次版本更新的一部分，Spring Batch 可运行在java8上。但它仍可在java6或更高版本(java6-java8)上运行。

# JobScope支持

在很长一段时间内，Spring Batch 的scope配置项“step”在批处理应用中起到关键作用，它提供了后期绑定功能。在Spring release 3.0版本，Spring Batch支持一个“job”的配置项。这个配置允许对象延迟创建，一般直到每个job将要执行时提供新的实例。你可在 Section 5.4.2, “Job Scope” 章节查看关于该配置的详细内容。

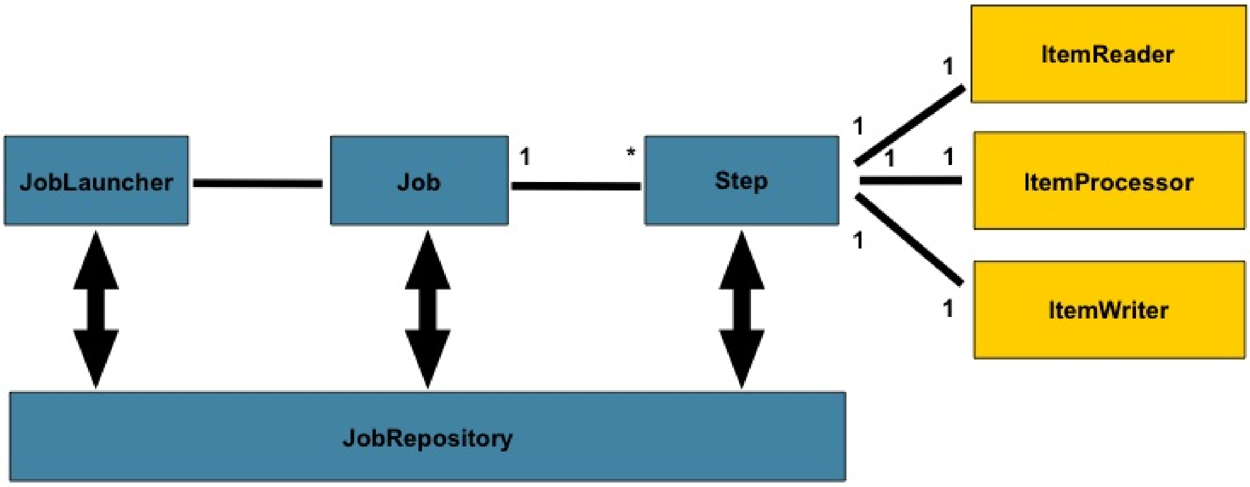
# SQLite支持

SQLite已作为新的数据库所支持，可为JobRepository添加job repository的DDL。这将提供了一个有用的、基于文件、数据存储的测试意图。

# 批处理的域语言

# 4. 配置并运行Job

在上一章节(domain section)，即批处理的域语言中,讨论了整体的架构设计，并使用如下关系图来进行表示：




虽然Job对象看上去像是对于多个Step的一个简单容器，但是开发者必须要注意许多配置项。此外，Job的运行以及Job运行过程中元数据如何被保存也是需要考虑的。本章将会介绍Job在运行时所需要注意的各种配置项。

# 4.1 Configuring a Job

Job接口 的实现有多个，但是在配置上命名空间存在着不同。必须依赖的只有三项：名称 **name**，**JobRespository** 和 **Step** 的列表：

```
<job id="footballJob">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s2" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
</job>
```

在这个例子中使用了父类的bean定义来创建step，更多描述step配置的信息可以参考step configuration这一节。XML命名空间默认会使用id为'jobRepository'的引用来作为repository的定义。然而可以向如下显式的覆盖：

```
<job id="footballJob" job-repository="specialRepository">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s3" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
</job>
```

此外，job配置的step还包含其他的元素，有并发处理()，显示的流程控制()和外化的流程定义()。

## 4.1.1 Restartablity

执行批处理任务的一个关键问题是要考虑job被重启后的行为。如果一个 **JobExecution** 已经存在一个特定的 **JobInstance** ，那么这个job启动时可以认为是"重启"。 理想情况下，所有任务都能够在他们中止的地方启动，但是有许多场景这是不可能的。在这种场景中就要有开发者来决定创建一个新的 **JobInstance** ， Spring对此也提供了一些帮助。如果job不需要重启，而是总是作为新的 **JobInstance** 来运行，那么可重启属性可以设置为'false'：

```
<job id="footballJob" restartable="false">
    ...
</job>
```

设置重启属性restartable为'false'表示'这个job不支持再次启动'，重启一个不可重启的job会抛出JobRestartException的异常：

```
Job job = new SimpleJob();
job.setRestartable(false);

JobParameters jobParameters = new JobParameters();

JobExecution firstExecution = jobRepository.createJobExecution(job, jobParameters);
jobRepository.saveOrUpdate(firstExecution);

try {
    jobRepository.createJobExecution(job, jobParameters);
    fail();
}
catch (JobRestartException e) {
    //预计抛出JobRestartException异常
}
```

这个JUnit代码展示了创建一个不可重启的Job后，第一次能够创建 **JobExecution** ， 第二次再创建相同的JobExcution会抛出一个 **JobRestartException**。

## 4.1.2 Intercepting Job Execution

在job执行过程中，自定义代码能够在生命周期中通过事件通知执行会是很有用的。SimpleJob能够在适当的时机调用JobListener：

```
public interface JobExecutionListener {

    void beforeJob(JobExecution jobExecution);

    void afterJob(JobExecution jobExecution);

}
```

JobListener能够添加到SimpleJob中去，作为job的listener元素：

```
<job id="footballJob">
    <step id="playerload"          parent="s1" next="gameLoad"/>
    <step id="gameLoad"            parent="s2" next="playerSummarization"/>
    <step id="playerSummarization" parent="s3"/>
    <listeners>
        <listener ref="sampleListener"/>
    </listeners>
</job>
```

无论job执行成功或是失败都会调用afterJob，都可以从 **JobExecution** 中获取运行结果后，根据结果来进行不同的处理：

```
public void afterJob(JobExecution jobExecution){
    if( jobExecution.getStatus() == BatchStatus.COMPLETED ){
        //job执行成功    }
    else if(jobExecution.getStatus() == BatchStatus.FAILED){
        //job执行失败    }
}
```

对应于这个interface的annotation为：

- @BeforeJob
- @AfterJob

## 4.1.3 Inheriting from a parent Job

如果一组job配置共有相似，但又不是完全相同，那么可以定义一个"父"job，让这些job去继承属性。同Java的类继承一样，子job会把父job的属性和元素合并进来。

下面的例子中，“baseJob”是一个抽象的job定义，只定义了一个监听器列表。名为“job1”的job是一个具体定义，它继承了“baseJob"的监听器，并且与自己的监听器合并，最终生成的job带有两个监听器，以及一个名为”step1“的step。

```
<job id="baseJob" abstract="true">
    <listeners>
        <listener ref="listenerOne"/>
    <listeners>
</job>

<job id="job1" parent="baseJob">
    <step id="step1" parent="standaloneStep"/>

    <listeners merge="true">
        <listener ref="listenerTwo"/>
```

```
    <listeners>
</job>
```

更多信息可参见 Inheriting from a Parent Step

## 4.1.4 JobParametersValidator

一个在xml命名空间描述的job或是使用任何抽象job子类的job，可以选择为运行时为job参数定义一个验证器。在job启动时需要保证所有必填参数都存在的场景下，这个功能是很有用的。有一个DefaultJobParametersValidator可以用来限制一些简单的必选和可选参数组合，你也可以实现接口用来处理更复杂的限制。验证器的配置支持使用xml命名空间来作为job的子元素，例如：

```
<job id="job1" parent="baseJob3">
    <step id="step1" parent="standaloneStep"/>
    <validator ref="paremetersValidator"/>
</job>
```

验证器可以作为一个引用(如上)来定义也可以直接内嵌定义在bean的命名空间中。

# 4.2 Java Config

在Spring 3版本中可以采用java程序来配置应用程序，来替代XML配置的方式。 正如在Spring Batch 2.2.0版本中，批处理任务中可以使用相同的java配置项来对其进行配置。关于Java的基础配置的两个组成部分分别是： `@EnableBatchConfiguration` 注释和两个builder。

在Spring的体系中 `@EnableBatchProcessing` 注释的工作原理与其它的带有 `@Enable *` 的注释类似。在这种情况下， `@EnableBatchProcessing` 提供了构建批处理任务的基本配置。在这个基本的配置中，除了创建了一个 **StepScope** 的实例，还可以将一系列可用的bean进行自动装配:

- **JobRepository** bean 名称 "jobRepository"
- **JobLauncher** bean名称"jobLauncher"
- **JobRegistry** bean名称"jobRegistry"
- **PlatformTransactionManager** bean名称 "transactionManager"
- **JobBuilderFactory** bean名称"jobBuilders"
- **StepBuilderFactory** bean名称"stepBuilders"

这种配置的核心接口是 **BatchConfigurer**。它为以上所述的bean提供了默认的实现方式，并要求在context中提供一个bean，即 **DataSource** 。数据库连接池由被 **JobRepository** 使用。

注意 只有一个配置类需要有@ enablebatchprocessing注释。只要有一个类添加了这个注释，则以上所有的bean都是可以使用的。

在基本配置中，用户可以使用所提供的builder factory来配置一个job。下面的例子是通过 **JobBuilderFactory** 和 **StepBuilderFactory** 配置的两个step job 。

```
@Configuration
@EnableBatchProcessing
@Import(DataSourceCnfiguration.class)
public class AppConfig {

    @Autowired
    private JobBuilderFactory jobs;

    @Autowired
    private StepBuilderFactory steps;

    @Bean
    public Job job() {
        return jobs.get("myJob").start(step1()).next(step2()).build();
    }

    @Bean
    protected Step step1(ItemReader<Person> reader, ItemProcessor<Person, Person> processor, ItemWriter<Person> writer)
        return steps.get("step1")
            .<Person, Person> chunk(10)
            .reader(reader)
            .processor(processor)
            .writer(writer)
            .build();
    }

    @Bean
    protected Step step2(Tasklet tasklet) {
        return steps.get("step2")
            .tasklet(tasklet)
            .build();
    }
}
```

# 4.3 Configuring a JobRepository

之前说过，**JobRepository** 是基本的CRUD操作，用于持久化Spring Batch的领域对象(如JobExecution,StepExecution)。许多主要的框架组件(如JobLauncher,Job,Step)都需要使用JobRepository。batch的命名空间中已经抽象走许多JobRepository的实现细节，但是仍然需要一些配置：

```
<job-repository id="jobRepository"
    data-source="dataSource"
    transaction-manager="transactionManager"
    isolation-level-for-create="SERIALIZABLE"
    table-prefix="BATCH_"
    max-varchar-length="1000"/>
```

上面列出的配置除了id外都是可选的。如果没有进行参数配置，默认值就是上面展示的内容，之所以写出来是用于展示给读者。 `max-varchar-length` 的默认值是2500，这表示varchar列的长度，在 sample schema scripts 中用于存储类似于 `exit code` 这些描述的字符。如果你不修改schema并且也不会使用多字节编码，那么就不用修改它。

## 4.3.1 JobRepository 的事物配置

如果使用了namespace，repository会被自动加上事务控制，这是为了确保批处理操作元数据以及失败后重启的状态能够被准确的持久化，如果repository的方法不是事务控制的，那么框架的行为就不能够被准确的定义。 `create*` 方法的隔离级别会被单独指定，为了确保任务启动时，如果两个操作尝试在同时启动相同的任务，那么只有一个任务能够被成功启动。这种方法默认的隔离级别是 `SERIALIZABLE` ，这是相当激进的做法： `READ_COMMITED` 能达到同样效果；如果两个操作不以这种方式冲突的话 `READ_UNCOMMITED` 也能很好工作。但是，由于调用 `create*` 方法是相当短暂的，只要数据库支持，就不会对性能产生太大影响。它也能被这样覆盖：

```
<job-repository id="jobRepository"
                isolation-level-for-create="REPEATABLE_READ" />
```

如果factory的namespace没有被使用，那么可以使用AOP来配置repository的事务行为：

```
<aop:config>
    <aop:advisor
           pointcut="execution(* org.springframework.batch.core..*Repository+.*(..))"/>
    <advice-ref="txAdvice" />
</aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="*" />
    </tx:attributes>
</tx:advice>
```

这个配置片段基本上可以不做修改直接使用。记住加上适当的namespace描述去确保spring-tx和spring-aop(或是整个spring)都在classpath中。

## 4.3.2 修改 Table 前缀

**JobRepository** 可以修改的另一个属性是元数据表的表前缀。默认是以BATCH_开头， `BATCH_JOB_EXECUTION` 和 `BATCH_STEP_EXECUTION` 就是两个例子。但是，有一些潜在的原因可能需要修改这个前缀。例如schema的名字需要被预置到表名中，或是不止一组的元数据表需要放在同一个schema中，那么表前缀就需要改变：

```
<job-repository id="jobRepository"
            table-prefix="SYSTEM.TEST_" />
```

按照上面的修改配置，每一个元数据查询都会带上 `SYSTEM.TEST_` 的前缀， `BATCH_JOB_EXECUTION` 将会被更换为 `SYSTEM.TEST_JOB_EXECUTION` 。

注意：表名前缀是可配置的，表名和列名是不可配置的。

## 4.3.3 In-Memory Repository

有的时候不想把你的领域对象持久化到数据库中，可能是为了运行的更快速，因为每次提交都要开销额外的时间；也可能并不需要为特定任务保存状态。那么Spring Batch还提供了内存Map版本的job仓库：

```
<bean id="jobRepository"
  class="org.springframework.batch.core.repository.support.MapJobRepositoryFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
</bean>
```

需要注意的是 内存 **Repository** 是轻量的并且不能在两个JVM实例间重启任务，也不能允许同时启动带有相同参数的任务，不适合在多线程的任务或是一个本地分片任务的场景下使用。而使用数据库版本的Repository则能够拥有这些特性。

但是也需要定义一个事务管理器，因为仓库需要回滚语义，也因为商业逻辑要求事务性（例如RDBMS访问）。经过测试许多人觉得 **ResourcelessTransactionManager** 是很有用的。

## 4.3.4 Non-standard Database Types in a Repository

如果使用的数据库平台不在支持的平台列表中，在SQL类型类似的情况下你可以使用近似的数据库类型。使用原生的 **JobRepositoryFactoryBean** 来取代命名空间缩写后设置一个相似的数据库类型：

```
<bean id="jobRepository" class="org...JobRepositoryFactoryBean">
    <property name="databaseType" value="db2"/>
    <property name="dataSource" ref="dataSource"/>
</bean>
```

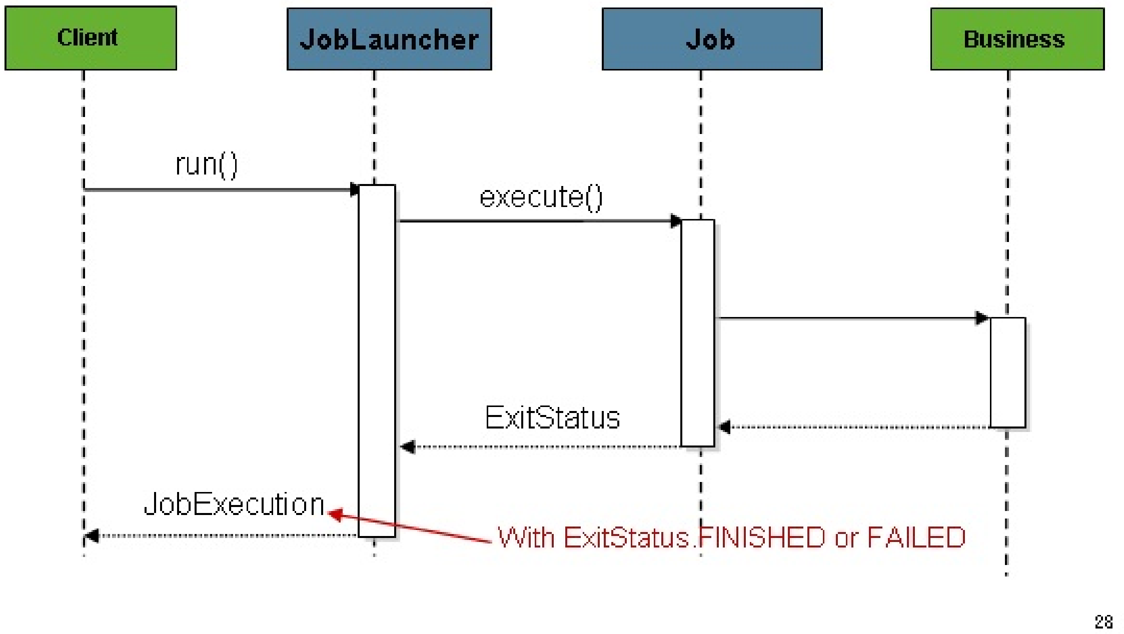
(如果没有指定 `databaseType`， **JobRepositoryFactoryBean** 会通过DataSource自动检测数据库的类型).平台之间的主要不同之处在于主键的计算策略，也可能需要覆盖 `incrementerFactory` (使用Spring Framework提供的标准实现)。 如果它还不能工作，或是你不使用RDBMS，那么唯一的选择是让 **SimpleJobRepository** 使用Spring方式依赖并且绑定在手工实现的各种Dao接口上。

# 4.4 Configuring a JobLauncher

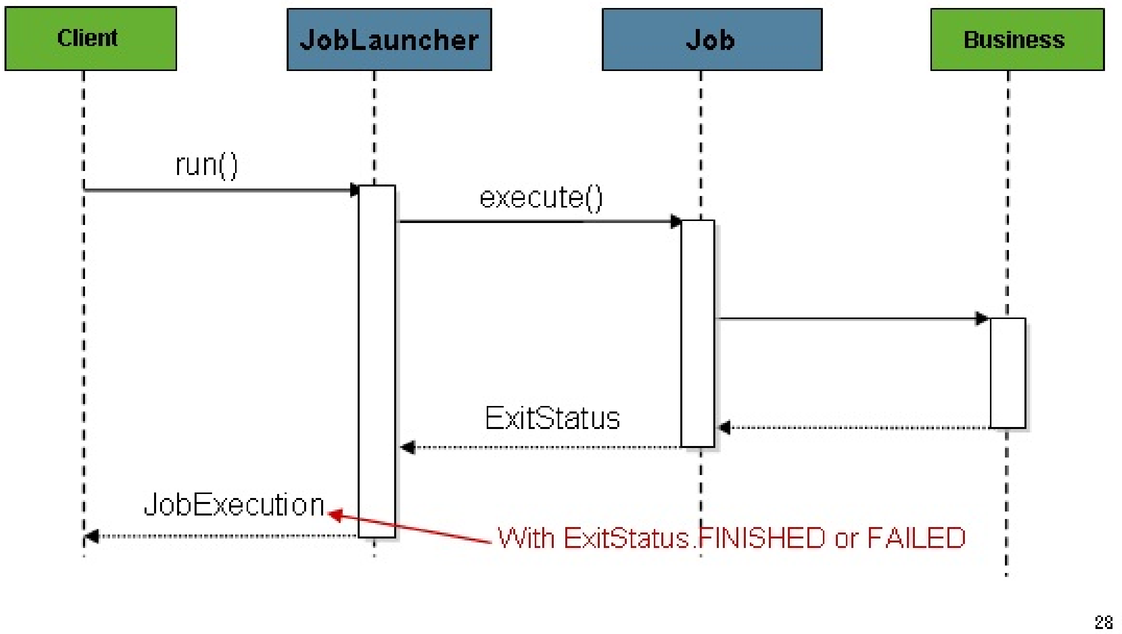
`JobLauncher` 最基本的实现是 `SimpleJobLauncher`，它唯一的依赖是通过 `JobRepository` 获取一个 `execution`：

```
<bean id="jobLauncher"
      class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
</bean>
```

一旦获取到 **JobExecution**，那么可以通过执行 `Job` 的方法，最终将 `JobExecution` 返回给调用者：




从调度启动时，整个序列能够很好的直接工作，但是，从HTTP请求中启动则会出现一些问题。在这种场景中，启动任务需要异步操作，让**SimpleJobLauncher**能够立刻返回结果给调用者，如果让HTTP请求一直等待很长时间知道批处理任务完成获取到执行结果，是很糟糕的操作体验。一个流程如下图所示：




通过配置 `TaskExecutor` 可以很容易的将 `SimpleJobLauncher` 配置成异步操作：

```
<bean id="jobLauncher"
      class="org.springframework.batch.core.launch.support.SimpleJobLauncher">
    <property name="jobRepository" ref="jobRepository" />
    <property name="taskExecutor">
        <bean class="org.springframework.core.task.SimpleAsyncTaskExecutor" />
    </property>
</bean>
```

**TaskExecutor** 接口的任何实现都能够用来控制 `job` 的异步执行。

# 4.5 Running a Job

运行一个批处理任务至少有两点要求：一个 `JobLauncher` 和一个用来运行的 `job` 。它们都包含了相同或是不同的 `context` 。举例来说，从命令行来启动job，会为每一个job初始化一个JVM，因此每个job会有一个自己的 **JobLauncher**；从web容器的HttpRequest来启动job，一般只是用一个 **JobLauncher** 来异步启动job，http请求会调用这个 **JobLauncher** 来启动它们需要的job。

## 4.5.1 从命令行启动 Jobs

对于使用企业级调度器来运行job的用户来说，命令行是主要的使用接口。这是因为大多数的调度器是之间工作在操作系统进程上(除了Quartz, 否则使用 `NativeJob` )，使用shell脚本来启动。除了shell脚本还有许多脚本语言能启动java进程，如Perl和Ruby，甚至一些构建工具也可以，如ant和maven。但是大多数人都熟悉shell脚本，这个例子主要展示shell脚本。

**The CommandLineJobRunner**

由于脚本启动job需要开启java虚拟机，那么需要有一个类带有main方法作为操作的主入口点。Spring Batch针对这个需求提供了一个实现： **CommandLineJobRunner** 。需要强调的是这只是引导你的应用程序的一种方法，有许多方法能够启动java进程，不应该把这个类视为最终方法。**CommandLineJobRunner**执行四个任务：

- 加载适当的 `ApplicationContext`
- 解析到 `JobParameters` 的命令行参数
- 根据参数确定合适的任务
- 使用在application context(应用上下文)中所提供的 `JobLauncher` 来启动job。

所有这些任务只要提供几个参数就可以完成。以下是要求的参数：

Table 4.1. CommandLineJobRunner arguments

| | |
|---|---|
| jobPath | 用于创建ApplicationContext的xml文件地址。这个文件包含了完成任务的一切配置。 |
| jobName | 需要运行的job的名字。 |

参数中必须路径参数在前，任务名参数在后。被设置到JobParameter中的参数必须使用" `name=value` "的格式：

```
bash$ java CommandLineJobRunner endOfDayJob.xml endOfDay schedule.date(date)=2007/05/05
```

大多数情况下在jar中放置一个manifest文件来描述main class，但是直接使用class会比较简洁。还是使用 domain section中的'EndOfDay'例子，第一个参数是'endOfDayJob.xml'，这是包含了Job和Spring ApplicationContext；第二个参数是'endOfDay'，指定了Job的名字；最后一个参数'schedule.date(date)=2007/05/05'会被转换成JobParameters。例子中的xml如下所示：

```
<job id="endOfDay">
    <step id="step1" parent="simpleStep" />
</job>

<!-为清晰起见省略了Launcher的详细信息-->
<beans:bean id="jobLauncher"
     class="org.springframework.batch.core.launch.support.SimpleJobLauncher" />
```

例子很简单，在实际案例中Spring Batch运行一个Job通常有多得多的要求，但是这里展示了 **CommandLineJobRunner** 的

两个主要要求：`Job` 和 `JobLauncher`。

**ExitCodes**

使用企业级调度器通过命令行启动一个批处理任务后，大多数调度器都是在进程级别沉默的工作。这意味着它们只知道一些操作系统进程信息(如它们执行的脚本)。在这种场景下，只能通过返回code来和调度器交流job执行成功还是失败的信息。返回code是返回给调度程序进程的一个数字，用于指示运行结果。最简单的一个例子：0表示成功，1表示失败。更复杂的场景如：job A返回4就启动job B，返回5就启动job C。这种类型的行为被配置在调度器层级，但重要的是像Spring Batch这种处理框架需要为特殊的批处理任务提供一个返回'Exit Code'数字表达式。在SpringBatch中，退出代码被封装在**ExitStatus**，具体细节会在Chapter 5中介绍。对于 `exit code`，只需要知道**ExitStatus**有一个 `exit code` 属性能够被框架或是开发者设置，作为**JobLauncher**返回的**JobExecution**的一部分。**CommandLineJobRunner** 使用 **ExitCodeMapper** 接口将字符串的值转换为数值：

```
public interface ExitCodeMapper {

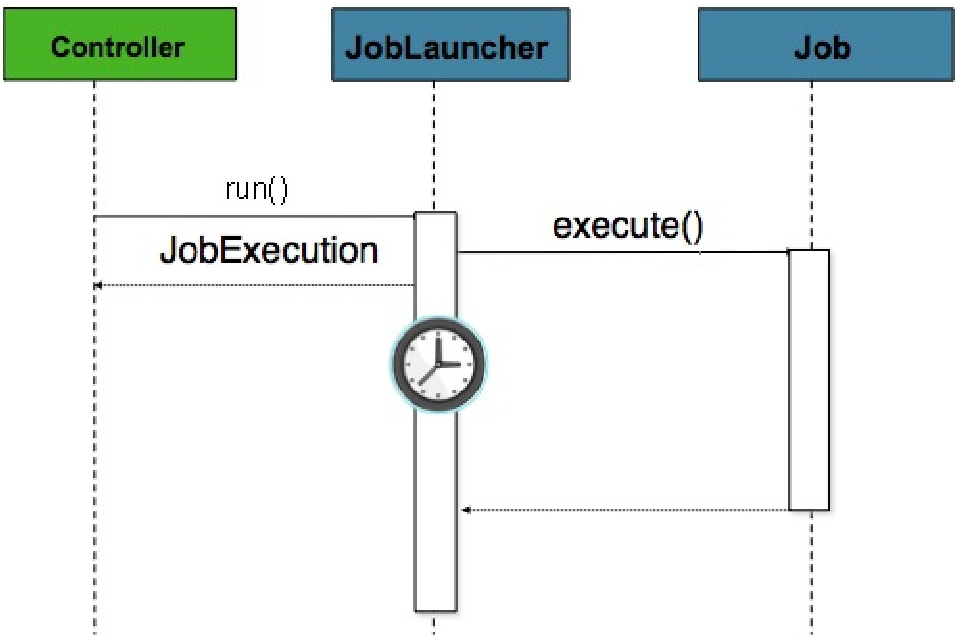
    public int intValue(String exitCode);

}
```

**ExitCodeMapper** 的基本协议是传入一个字符串，返回一个数字表达式。job运行器默认使用的是 **SimpleJvmExitCodeMapper**，完成返回0，一般错误返回1，上下文找不到**job**，这类job运行器启动级别的错误则返回2。如果需要比上面三个值更复杂的返回值，就提供自定义 **ExitCodeMapper** 的实现。由于 **CommandLineJobRunner** 是创建 **ApplicationContext** 的类，不能够使用绑定功能，所以所有的值都需要覆盖后使用自动绑定，因此 **ExitCodeMapper** 在 **BeanFactory** 中加载，就会在上下文被创建后注入到job运行器中。而所有需要做的就是提供自己的 **ExitCodeMapper** 描述为 **ApplicationContext** 的一部分，使之能够被运行器加载。

## 4.5.2 在 Web Container 内部运行 Jobs

过去，像批处理任务这样的离线计算都需要从命令行启动。但是，许多例子(包括报表、点对点任务和web支持)都表明，从HttpRequest启动是一个更好的选择。另外，批处理任务一般都是需要长时间运行，异步启动时最为重要的：




这个例子中的Controller就是spring MVC中的Controller(Spring MVC的信息可以在 http://docs.spring.io/spring/docs/3.2.x/spring-framework-reference/html/mvc.html 中查看)。Controller通过使用配置为异步的(asynchronously)JobLauncher启动job后立即返回了JobExecution。job保持运行，这个非阻塞的行为能够让controller在持有

HttpRequest时立刻返回。示例如下：

```
@Controller
public class JobLauncherController {

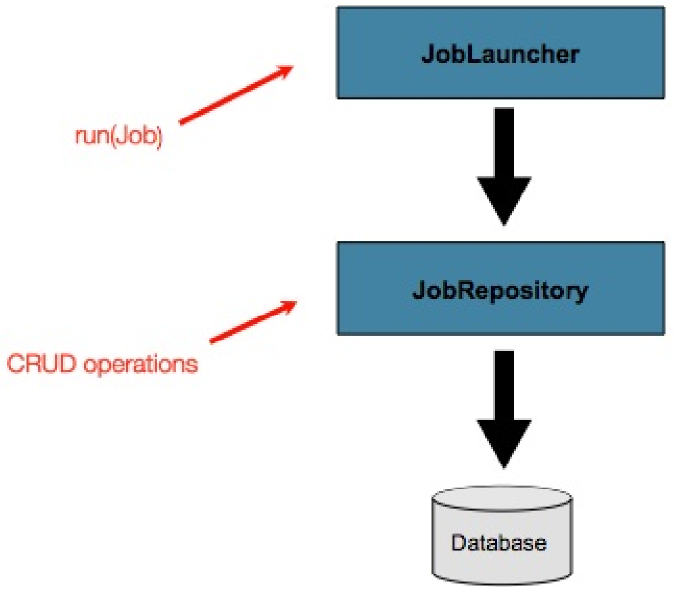
    @Autowired
    JobLauncher jobLauncher;

    @Autowired
    Job job;

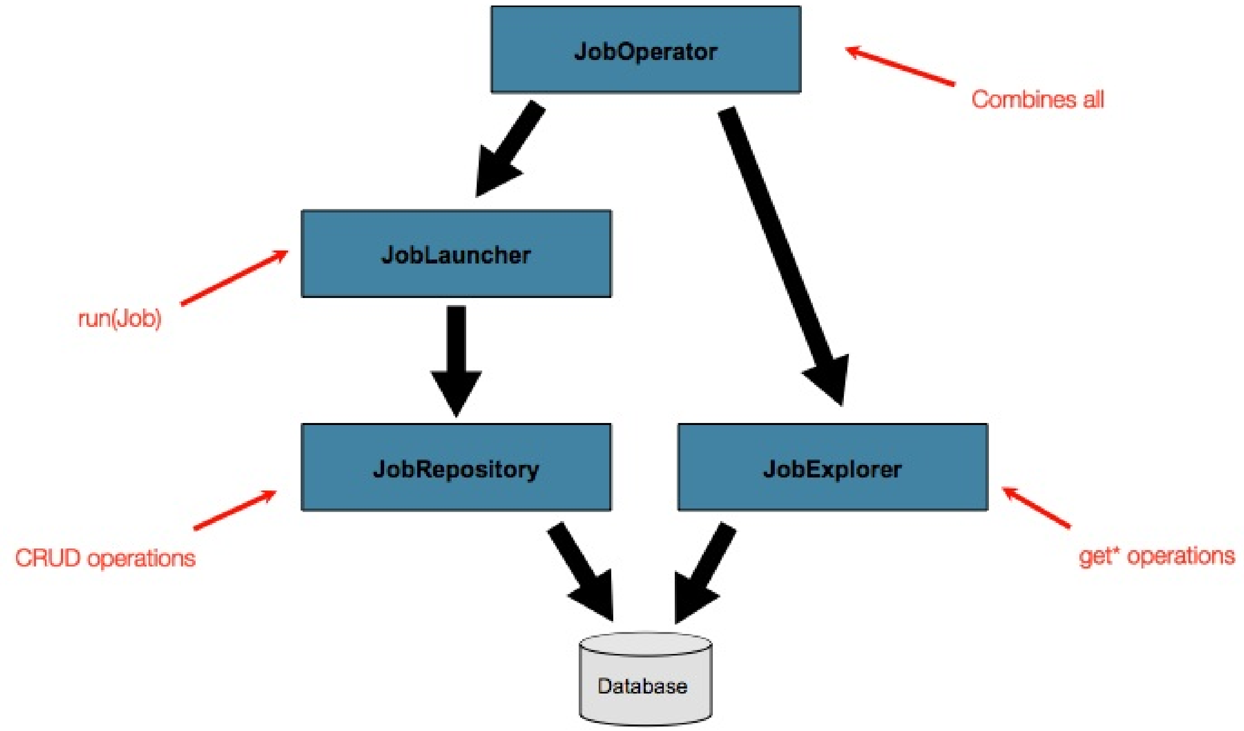
    @RequestMapping("/jobLauncher.html")
    public void handle() throws Exception{
        jobLauncher.run(job, new JobParameters());
    }
}
```

# 4.6 Meta-Data 高级用法

到目前为止，已经讨论了 **JobLauncher** 和 **JobRepository** 接口，它们展示了简单启动任务，以及批处理领域对象的基本CRUD操作：




一个JobLauncher使用一个JobRepository创建并运行新的JobExection对象，Job和Step实现随后使用相同的JobRepository在job运行期间去更新相同的JobExecution对象。这些基本的操作能够满足简单场景的需要，但是对于有着数百个任务和复杂定时流程的大型批处理情况来说，就需要使用更高级的方式访问元数据：




接下去会讨论 **JobExplorer** 和 **JobOperator** 两个接口，能够使用更多的功能去查询和修改元数据。

## 4.6.1 Querying the Repository

在使用高级功能之前，需要最基本的方法来查询repository去获取已经存在的 `execution` 。**JobExplored** 接口提供了这些功能：

```
public interface JobExplorer {

    List<JobInstance> getJobInstances(String jobName, int start, int count);

    JobExecution getJobExecution(Long executionId);

    StepExecution getStepExecution(Long jobExecutionId, Long stepExecutionId);

    JobInstance getJobInstance(Long instanceId);

    List<JobExecution> getJobExecutions(JobInstance jobInstance);

    Set<JobExecution> findRunningJobExecutions(String jobName);
}
```

上面的代码表示的很明显，JobExplorer是一个只读版的JobRepository，同JobRepository一样，它也能够很容易配置一个工厂类：

```
<bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
      p:dataSource-ref="dataSource" />
```

（Earlier in this chapter）之前有提到过，**JobRepository** 能够配置不同的表前缀用来支持不同的版本或是schema。**JobExplorer** 也支持同样的特性：

```
<bean id="jobExplorer" class="org.spr...JobExplorerFactoryBean"
      p:dataSource-ref="dataSource" p:tablePrefix="BATCH_" />
```

## 4.6.2 JobRegistry

`JobRegistry` (父接口为 `JobLocator` )并非强制使用，它能够协助用户在上下文中追踪job是否可用，也能够在应用上下文收集在其他地方(子上下文)创建的job信息。自定义的JobRegistry实现常被用于操作job的名称或是其他属性。框架提供了一个基于map的默认实现，能够从job的名称映射到job的实例：

```
<bean id="jobRegistry" class="org.spr...MapJobRegistry" />
```

有两种方法自动注册job进JobRegistry：使用bean的post处理器或是使用注册生命周期组件。这两种机制在下面描述。

**JobRegistryBeanPostProcessor**

这是post处理器，能够将job在创建时自动注册进JobRegistry：

```
<bean id="jobRegistryBeanPostProcessor" class="org.spr...JobRegistryBeanPostProcessor">
    <property name="jobRegistry" ref="jobRegistry"/>
</bean>
```

并不一定要像例子中给post处理器一个id，但是使用id可以在子context中(比如作为作为父 bean 定义)也使用post处理器，这样所有的job在创建时都会自动注册进JobRegistry。

**AutomaticJobRegistrar**

这是生命周期组件，用于创建子context以及注册这些子context中的job。这种做法有一个好处，虽然job的名字仍然要求全局唯一，但是job的依赖项可以不用全局唯一，它可以有一个“自然”的名字。例如，创建了一组xml配置文件，每个文件有一个job，每个job的ItemReader都有一个相同的名字(如"reader")，如果这些文件被导入到一个上下文中，reader的定义会冲突并且互相覆盖。如果使用了自动注册机就能避免这一切发生。这样集成几个不同的应用模块就变得更容易了：

```
<bean class="org.spr...AutomaticJobRegistrar">
   <property name="applicationContextFactories">
      <bean class="org.spr...ClasspathXmlApplicationContextsFactoryBean">
         <property name="resources" value="classpath*:/config/job*.xml" />
      </bean>
   </property>
   <property name="jobLoader">
      <bean class="org.spr...DefaultJobLoader">
         <property name="jobRegistry" ref="jobRegistry" />
      </bean>
   </property>
</bean>
```

注册机有两个主要的属性，一个是ApplicationContextFactory数组(这儿创建了一个简单的factory bean)，另一个是 `jobLoader`。**JobLoader** 负责管理子context的生命周期以及注册任务到JobRegistry。

**ApplicationContextFactory** 负责创建子 **Context**，大多数情况下像上面那样使用 **ClassPathXmlApplicationContextFactory**。这个工厂类的一个特性是默认情况下他会复制父上下文的一些配置到子上下文。因此如果不变的情况下不需要重新定义子上下文中的 **PropertyPlaceholderConfigurer** 和AOP配置。

在必要情况下，**AutomaticJobRegistrar** 可以和 **JobRegistyBeanPostProcessor** 一起使用。例如，job有可能既定义在父上下文中也定义在子上下文中的情况。

## 4.6.3 JobOperator

正如前面所讨论的，JobRepository 提供了对元数据的 CRUD 操作，JobExplorer 提供了对元数据的只读操作。然而，这些操作最常用于联合使用诸多的批量操作类，来对任务进行监测，并完成相当多的任务控制功能，比如停止、重启或对任务进行汇总。在Spring Batch 中JobOperator 接口提供了这些操作类型：

```
public interface JobOperator {

    List<Long> getExecutions(long instanceId) throws NoSuchJobInstanceException;

    List<Long> getJobInstances(String jobName, int start, int count)
          throws NoSuchJobException;

    Set<Long> getRunningExecutions(String jobName) throws NoSuchJobException;

    String getParameters(long executionId) throws NoSuchJobExecutionException;

    Long start(String jobName, String parameters)
          throws NoSuchJobException, JobInstanceAlreadyExistsException;

    Long restart(long executionId)
          throws JobInstanceAlreadyCompleteException, NoSuchJobExecutionException,
                  NoSuchJobException, JobRestartException;

    Long startNextInstance(String jobName)
          throws NoSuchJobException, JobParametersNotFoundException, JobRestartException,
                 JobExecutionAlreadyRunningException, JobInstanceAlreadyCompleteException;

    boolean stop(long executionId)
          throws NoSuchJobExecutionException, JobExecutionNotRunningException;

    String getSummary(long executionId) throws NoSuchJobExecutionException;
```

```
    Map<Long, String> getStepExecutionSummaries(long executionId)
          throws NoSuchJobExecutionException;

    Set<String> getJobNames();

}
```

上图中展示的操作重现了来自其它接口提供的方法，比如JobLauncher, JobRepository, JobExplorer, 以及 JobRegistry。因为这个原因，所提供的JobOperator的实现SimpleJobOperator的依赖项有很多：

```
<bean id="jobOperator" class="org.spr...SimpleJobOperator">
    <property name="jobExplorer">
        <bean class="org.spr...JobExplorerFactoryBean">
            <property name="dataSource" ref="dataSource" />
        </bean>
    </property>
    <property name="jobRepository" ref="jobRepository" />
    <property name="jobRegistry" ref="jobRegistry" />
    <property name="jobLauncher" ref="jobLauncher" />
</bean>
```

注意 如果你在JobRepository中设置了表前缀，那么不要忘记在JobExplorer中也做同样设置。

# 4.6.4 JobParametersIncrementer

**JobOperator** 的多数方法都是不言自明的，更多详细的说明可以参见该接口的javadoc(javadoc of the interface)。然而startNextInstance方法却有些无所是处。这个方法通常用于启动Job的一个新的实例。但如果 JobExecution 存在若干严重的问题，同时该Job 需要从头重新启动，那么这时候这个方法就相当有用了。不像JobLauncher ，启动新的任务时如果参数不同于任何以往的参数集，这就要求一个新的 JobParameters 对象来触发新的 JobInstance，startNextInstance 方法将使用当前的JobParametersIncrementer绑定到这个任务，并强制其生成新的实例：

```
public interface JobParametersIncrementer {
    JobParameters getNext(JobParameters parameters);
}
```

**JobParametersIncrementer** 的协议是这样的，当给定一个 JobParameters 对象，它将返回填充了所有可能需要的值 “下一个” JobParameters 对象。这个策略非常有用，因为框架无需知晓变成“下一个”的JobParameters 做了哪些更改。例如，如果任务参数中只包含一个日期参数，那么当创建下一个实例时，这个值就应该是不是该自增一天？或者一周（如果任务是以周为单位运行的话）？任何包含数值类参数的任务，如果需要对其进行区分，都涉及这个问题，如下：

```
public class SampleIncrementer implements JobParametersIncrementer {

    public JobParameters getNext(JobParameters parameters) {
        if (parameters==null || parameters.isEmpty()) {
            return new JobParametersBuilder().addLong("run.id", 1L).toJobParameters();
        }
        long id = parameters.getLong("run.id",1L) + 1;
        return new JobParametersBuilder().addLong("run.id", id).toJobParameters();
    }
}
```

在该示例中，键值“`run.id`”用以区分各个JobInstance。如果当前的JobParameters为空(null)，它将被视为该Job从未运行过，并同时为其初始化，然后返回。反之，非空的时候自增一个数值，再返回。自增的数值可以在命名空间描述中通过Job的“incrementer”属性进行设置：

```
<job id="footballJob" incrementer="sampleIncrementer">
```

```
    ...
</job>
```

# 4.6.5 Stopping a Job

**JobOperator** 最常见的作用莫过于停止某个Job：

```
Set<Long> executions = jobOperator.getRunningExecutions("sampleJob");
jobOperator.stop(executions.iterator().next());
```

关闭不是立即发生的，因为没有办法将一个任务立刻强制停掉，尤其是当任务进行到开发人员自己的代码段时，框架在此刻是无能为力的，比如某个业务逻辑处理。而一旦控制权还给了框架，它会立刻设置当前 `StepExecution` 为 `BachStatus.STOPPED`， 意为停止，然后保存，最后在完成前对**JobExecution**进行相同的操作。

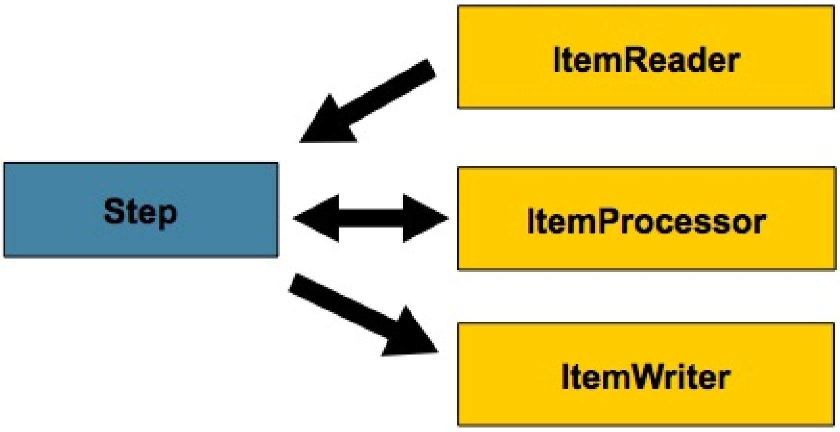
# 4.6.6 Aborting a Job

一个job的执行过程当执行到FAILED状态之后，如果它是可重启的，它将会被重启。如果任务的执行过程状态是ABANDONED，那么框架就不会重启它。ABANDONED状态也适用于执行步骤，使得它们可以被跳过，即便是在一个可重启的任务执行之中：如果任务执行过程中碰到在上一次执行失败后标记为ABANDONED的步骤，将会跳过该步骤直接到下一步(这是由任务流定义和执行步骤的退出码决定的)。

如果当前的系统进程死掉了(“kill -9”或系统错误)，job自然也不会运行，但JobRepository是无法侦测到这个错误的，因为进程死掉之前没有对它进行任何通知。你必须手动的告诉它，你知道任务已经失败了还是说考虑放弃这个任务（设置它的状态为FAILED或ABANDONED）-这是业务逻辑层的事情，无法做到自动决策。只有在不可重启的任务中才需要设置为FAILED状态，或者你知道重启后数据还是有效的。Spring Batch Admin中有一系列工具JobService，用以取消正在进行执行的任务。

# 配置Step

正如在Batch Domain Language中叙述的，Step是一个独立封装域对象，包含了所有定义和控制实际处理信息批任务的序列。这是一个比较抽象的描述，因为任意一个Step的内容都是开发者自己编写的Job。一个Step的简单或复杂取决于开发者的意愿。一个简单的Step也许是从本地文件读取数据存入数据库，写很少或基本无需写代码。一个复杂的Step也许有复杂的业务规则（取决于所实现的方式），并作为整个个流程的一部分。



# ItemReaders和ItemWriters

所有的批处理都可以描述为最简单的形式： 读取大量的数据, 执行某种类型的计算/转换, 以及写出执行结果. Spring Batch 提供了三个主要接口来辅助执行大量的读取与写出: **ItemReader**, **ItemProcessor** 和 **ItemWriter**.

# 6.1 ItemReader

最简单的概念, **ItemReader** 就是一种从各个输入源读取数据,然后提供给后续步骤的方式. 最常见的例子包括:

- **Flat** FileFlat File Item Readers 从纯文本文件中读取一行行的数据, 存储数据的纯文本文件通常具有固定的格式, 并且使用某种特殊字符来分隔每条记录中的各个字段(例如逗号,Comma).
- **XML** XML ItemReaders 独立地处理XML,包括用于解析、映射和验证对象的技术。还可以对输入数据的XML文件执行XSD schema验证。
- **Database** 数据库就是对请求返回结果集的资源,结果集可以被映射转换为需要处理的对象。默认的SQL ItemReaders调用一个 `RowMapper` 来返回对象, 并跟踪记录当前行,以备有重启的情况, 存储基本统计信息,并提供一些事务增强特性,关于事物将在稍后解释。

虽然有各种各样的数据输入方式, 但本章我们只关注最基本的部分。关于详细的可用 ItemReaders 列表可以参照 附录A

**ItemReader** 是一个通用输入操作的基本接口:

```
public interface ItemReader<T> {

    T read() throws Exception, UnexpectedInputException, ParseException;

}
```

`read` 是**ItemReader**中最根本的方法; 每次调用它都会返回一个 Item 或 null(如果没有更多item)。每个 item条目, 一般对应文件中的一行(line), 或者对应数据库中的一行(row), 也可以是XML文件中的一个元素(element)。 一般来说, 这些item都可以被映射为一个可用的domain对象(如 Trade, User 等等), 但也不是强制要求(最偷懒的方式,返回一个Map)。

一般约定 **ItemReader** 接口的实现都是向前型的(forward only). 但如果底层资源是事务性质的(如JMS队列),并且发生回滚(rollback), 那么下一次调用 `read` 方法有可能会返回和前次逻辑上相等的结果(对象)。值得一提的是, 处理过程中如果没有items, **ItemReader** 不应该抛出异常。例如,数据库 **ItemReader** 配置了一条查询语句, 返回结果数为0, 则第一次调用**read**方法将返回null。

# 6.2 ItemWriter

**ItemWriter** 在功能上类似于 **ItemReader**,但属于相反的操作。 资源仍然需要定位,打开和关闭, 区别就在于在于**ItemWriter** 执行的是写入操作(write out), 而不是读取。 在使用数据库或队列的情况下,写入操作对应的是插入( `insert` ),更新( `update` ),或发送( `send` )。 序列化输出的格式依赖于每个批处理作业自己的定义。

和 **ItemReader** 接口类似, **ItemWriter** 也是个相当通用的接口:

```
public interface ItemWriter<T> {

    void write(List<? extends T> items) throws Exception;

}
```

类比于**ItemReader**中的 `read` ， `write` 方法是**ItemWriter** 接口的根本方法; 只要传入的 items 列表是打开的,那么它就会尝试着将其写入(write out)。 因为一般来说, items 将要被批量写入到一起,然后再输出, 所以 `write` 方法接受一个List 参数,而不是单个对象(item)。 list 被输出后, 在 `write` 方法返回(return)之前, 对缓冲执行刷出(flush)操作是很必要的。 例如, 如果使用 Hibernate DAO 时, 对每个对象要调用一次DAO写操作, 操作完成之后, 方法 return 之前, writer 就应该关闭 hibernate 的 Session会话。

# 6.3 ItemProcessor

**ItemReader** 和 **ItemWriter** 接口对于每个任务来说都是非常必要的, 但如果想要在写出数据之前执行某些业务逻辑操作时要怎么办呢? 一个选择是对读取(reading)和写入(writing)使用组合模式(composite pattern): 创建一个 **ItemWriter** 的子类实现, 内部包含另一个 **ItemWriter** 对象的引用(对于 **ItemReader** 也是类似的). 示例如下:

```
public class CompositeItemWriter<T> implements ItemWriter<T> {

    ItemWriter<T> itemWriter;

    public CompositeItemWriter(ItemWriter<T> itemWriter) {
        this.itemWriter = itemWriter;
    }

    public void write(List<? extends T> items) throws Exception {
        // ... 此处可以执行某些业务逻辑
       itemWriter.write(item);
    }

    public void setDelegate(ItemWriter<T> itemWriter){
        this.itemWriter = itemWriter;
    }
}
```

上面的类中包含了另一个**ItemWriter**引用,通过代理它来实现某些业务逻辑。 这种模式对于 **ItemReader** 也是一样的道理, 但也可能持有内部 **ItemReader** 所拥有的多个数据输入对象的引用。 在**ItemWriter**中如果我们想要自己控制 `write` 的调用也可能需要持有其他引用。

但假如我们只想在对象实际被写入之前 "改造" 一下传入的item, 就没必要实现**ItemWriter**和执行 **write** 操作: 我们只需要这个将被修改的item对象而已。 对于这种情况, Spring Batch提供了 **ItemProcessor** 接口:

```
public interface ItemProcessor<I, O> {

    O process(I item) throws Exception;
}
```

**ItemProcessor** 非常简单; 传入一个对象,对其进行某些处理/转换,然后返回另一个对象(也可以是同一个)。 传入的对象和返回的对象类型可以一样，也可以不一致。 关键点在于处理过程中可以执行一些业务逻辑操作,当然这完全取决于开发者怎么实现它。 一个**ItemProcessor**可以被直接关联到某个 Step(步骤),例如,假设**ItemReader** 的返回类型是 Foo 【译者注: Foo, Bar一类的词 就和BalaBala一样,没什么实际意义】,而在写出之前需要将其转换成类型Bar 的对象。 就可以编写一个**ItemProcessor**来执行这种转换:

```
public class Foo {}

public class Bar {
    public Bar(Foo foo) {}
}

public class FooProcessor implements ItemProcessor<Foo,Bar>{
    public Bar process(Foo foo) throws Exception {
        //执行某些操作,将 Foo 转换为 Bar对象
        return new Bar(foo);
    }
}

public class BarWriter implements ItemWriter<Bar>{
    public void write(List<? extends Bar> bars) throws Exception {
        //write bars
    }
}
```

在上面的简单示例中,有两个类: **Foo** 和 **Bar**, 以及实现了 **ItemProcessor** 接口的**FooProcessor**类。 因为是demo,所以转换很简单, 在实际使用中可能执行转换为任何类型, 响应的操作请读者根据需要自己编写。 **BarWriter**将被用于写出**Bar**对象,如果传入其他类型的对象可能会抛出异常。 同样,如果 **FooProcessor** 传入的参数不是 **Foo** 也会抛出异常。**FooProcessor**可以注入到某个**Step**中:

```
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="fooProcessor" writer="barWriter"
                   commit-interval="2"/>
        </tasklet>
    </step>
</job>
```

## 6.3.1 Chaining ItemProcessors

在很多情况下执行单个转换就可以了, 但假如想要将多个 **ItemProcessors** "串联(chain)" 在一起要怎么实现呢? 我们可以使用前面提到的组合模式(composite pattern)来完成。 接着前面单一转换的示例, 我们将**Foo**转换为**Bar**,然后再转换为**Foobar**类型,并执行写出:

```
public class Foo {}

public class Bar {
    public Bar(Foo foo) {}
}

public class Foobar{
    public Foobar(Bar bar) {}
}

public class FooProcessor implements ItemProcessor<Foo,Bar>{
    public Bar process(Foo foo) throws Exception {
        //Perform simple transformation, convert a Foo to a Bar
        return new Bar(foo);
    }
}

public class BarProcessor implements ItemProcessor<Bar,FooBar>{
    public FooBar process(Bar bar) throws Exception {
        return new Foobar(bar);
    }
}

public class FoobarWriter implements ItemWriter<FooBar>{
    public void write(List<? extends FooBar> items) throws Exception {
        //write items
    }
}
```

可以将 **FooProcessor** 和 **BarProcessor** “串联”在一起来生成 **Foobar** 对象,如果用 Java代码表示,那就像下面这样:

```
CompositeItemProcessor<Foo,Foobar> compositeProcessor = new  CompositeItemProcessor<Foo,Foobar>();
List itemProcessors = new ArrayList();
itemProcessors.add(new FooTransformer());
itemProcessors.add(new BarTransformer());
compositeProcessor.setDelegates(itemProcessors);
```

就和前面的示例类似,复合处理器也可以配置到**Step**中:

```
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="compositeProcessor" writer="foobarWriter"
                   commit-interval="2"/>
        </tasklet>
    </step>
</job>

<bean id="compositeItemProcessor"
      class="org.springframework.batch.item.support.CompositeItemProcessor">
    <property name="delegates">
        <list>
            <bean class="..FooProcessor" />
            <bean class="..BarProcessor" />
        </list>
    </property>
</bean>
```

## 6.3.2 Filtering Records

item processor 的典型应用就是在数据传给ItemWriter之前进行过滤(filter out)。 过滤(Filtering)是一种有别于跳过(skipping)的行为; skipping表明某几行记录是无效的,而 filtering 则只是表明某条记录不应该写入(written)。

例如, 某个批处理作业,从一个文件中读取三种不同类型的记录: 准备 insert 的记录、准备 update 的记录,需要 delete 的记录。如果系统中不允许删除记录, 那么我们肯定不希望将 “delete” 类型的记录传递给 **ItemWriter**。 但因为这些记录又不是损坏的信息(bad records), 我们只想将其过滤掉,而不是跳过。 因此,ItemWriter只会收到 "insert" 和 "update"的记录。

要过滤某条记录, 只需要 **ItemProcessor** 返回“`null`” 即可. 框架将自动检测结果为“`null`”的情况, 不会将该item 添加到传给 **ItemWriter**的list中。 像往常一样, 在 **ItemProcessor** 中抛出异常将会导致跳过(skip)。

## 6.3.3 容错(Fault Tolerance)

当某一个分块回滚时, 读取后已被缓存的那些item可能会被重新处理。 如果一个step被配置为支持容错(通常使用 skip跳过 或 retry重试处理),使用的所有 `ItemProcessor` 都应该实现为幂等的(idempotent)。 通常ItemProcessor对已经处理过的输入数据不执行任何修改, 而只更新需要处理的实例。

# 6.4 ItemStream

**ItemReader** 和 **ItemWriter** 都为各自的目的服务, 但他们之间有一个共同点, 就是都需要与另一个接口配合。 一般来说,作为批处理作业作用域范围的一部分,readers 和 writers 都需要打开(open),关闭(close),并需要某种机制来持久化自身的状态:

```
public interface ItemStream {

    void open(ExecutionContext executionContext) throws ItemStreamException;

    void update(ExecutionContext executionContext) throws ItemStreamException;

    void close() throws ItemStreamException;
}
```

在描述每种方法之前, 我们应该提到**ExecutionContext**。 **ItemReader**的客户端也应该实现 **ItemStream**, 在任何 `read` 之前调用 `open` 以打开需要的文件或数据库连接等资源。 实现**ItemWriter**也有类似的限制/约束,即需要同时实现**ItemStream**。 如第2章所述,如果将数据存放在**ExecutionContext** 中, 那么它可以在某个时刻用来启动 **ItemReader** 或 **ItemWriter**,而不是在初始状态时。 对应的, 应该确保在调用 `open` 之后的适当位置调用 `close` 来安全地释放所有分配的资源。 调用 `update` 主要是为了确保当前持有的所有状态都被加载到所提供的 **ExecutionContext**中。 `update` 一般在提交之前调用, 以确保当前状态被持久化到数据库之中。

在特殊情况下, **ItemStream** 的客户端是一个 **Step**(由 Spring Batch Core 决定), 会为每个 **StepExecution** 创建一个 **ExecutionContext**, 以允许用户存储特定部分的执行状态, 一般来说如果同一个**JobInstance**重启了,则预期它将会在重启后被返回。 对于熟悉 Quartz的人来说, 逻辑上非常像是 Quartz 的 **JobDataMap**。

# 6.5 委托模式(Delegate Pattern)与注册Step

请注意, **CompositeItemWriter**是委托模式的一个示例, 这在Spring Batch中很常见的。 委托自身可以实现回调接口 **StepListener**。 如果实现了,那么他们就会被当作**Job**中**Step**的一部分与 Spring Batch Core 结合使用, 然后他们基本上必定需要手动注册到 **Step** 中。

一个 reader, writer, 或 processor,如果实现了 **ItemStream** / **StepListener**接口,就会被自动组装到 Step 中。 但因为 delegates 并不为 **Step** 所知, 因此需要被注入(作为listeners监听器或streams流,或两者都可):

```
<job id="ioSampleJob">
    <step name="step1">
        <tasklet>
            <chunk reader="fooReader" processor="fooProcessor" writer="compositeItemWriter"
                   commit-interval="2">
                <streams>
                    <stream ref="barWriter" />
                </streams>
            </chunk>
        </tasklet>
    </step>
</job>

<bean id="compositeItemWriter" class="...CustomCompositeItemWriter">
    <property name="delegate" ref="barWriter" />
</bean>

<bean id="barWriter" class="...BarWriter" />
```

# 6.6 纯文本平面文件(Flat Files)

最常见的批量数据交换机制是使用纯文本平面文件(flat file)。 XML由统一约定好的标准来定义文件结构(即XSD), 与XML等格式不同, 想要阅读纯文本平面文件必须先了解其组成结构。一般来说,纯文本平面文件分两种类型: 有分隔的类型(Delimited) 与固定长度类型(Fixed Length)。有分隔的文件中各个字段由分隔符进行间隔, 比如英文逗号(,)。而固定长度类型的文件每个字段都有固定的长度。

## 6.6.1 The FieldSet(字段集)

当在Spring Batch中使用纯文本文件时, 不管是将其作为输入还是输出, 最重要的一个类就是 **FieldSet**。许多架构和类库会抽象出一些方法/类来辅助你从文件读取数据, 但是这些方法通常返回 `String` 或者 `String[]` 数组, 很多时候这确实是些半成品。 而 **FieldSet** 是Spring Batch中专门用来将文件绑定到字段的抽象。它允许开发者和使用数据库差不多的方式来使用数据输入文件入。 `FieldSet` 在概念上非常类似于Jdbc的 `ResultSet` 。 FieldSet 只需要一个参数: 即token数组 `String[]` 。另外, 您还可以配置字段的名称, 然后就可以像使用 `ResultSet` 一样, 使用 index 或者 name 都可以取得对应的值:

```
String[] tokens = new String[]{"foo", "1", "true"};
FieldSet fs = new DefaultFieldSet(tokens);
String name = fs.readString(0);
int value = fs.readInt(1);
boolean booleanValue = fs.readBoolean(2);
```

在 **FieldSet** 接口可以返回很多类型的对象/数据, 如 `Date` , `long` , `BigDecimal` 等。 FieldSet 最大的优势在于,它对文本输入文件 提供了统一的解析。 不是每个批处理作业采用不同的方式进行解析,而一直是一致的, 不论是在处理格式异常引起的错误, 还是在进行简单的数据转换。

## 6.6.2 FlatFileItemReader

> 译注:
>
> 本文中 将 Flat File 翻译为“平面文件”, 这是一种没有特殊格式的非二进制的文件，里面的内容没有相对关系结构的记录。

平面文件(flat file)是最多包含二维(表格)数据的任意类型的文件。在 Spring Batch 框架中 **FlatFileItemReader** 类负责读取平面文件, 该类提供了用于读取和解析平面文件的基本功能。FlatFileItemReader 主要依赖两个东西: **Resource** 和 **LineMapper**。LineMapper接口将在下一节详细讨论。 `resource` 属性代表一个 Spring Core Resource(Spring核心资源)。关于如何创建这一类 bean 的文档可以参考 Spring框架, Chapter 5.Resources。所以本文档就不再深入讲解创建 Resource 对象的细节。 但可以找到一个文件系统资源的简单示例，如下所示:

```
Resource resource = new FileSystemResource("resources/trades.csv");
```

在复杂的批处理环境中，目录结构通常由EAI基础设施管理, 并且会建立放置区(drop zones)，让外部接口将文件从ftp移动到批处理位置, 反之亦然。文件移动工具(File moving utilities)超出了spring batch架构的范畴, 但在批处理作业中包括文件移动步骤这种事情那也是很常见的。 批处理架构只需要知道如何定位需要处理的文件就足够了。Spring Batch 将会从这个起始点开始，将数据传输给数据管道。当然, Spring Integration也提供了很多这一类的服务。

**FlatFileItemReader** 中的其他属性让你可以进一步指定数据如何解析:

**Table 6.1. FlatFileItemReader 的属性(Properties)**

| 属性(Property) | 类型(Type) | 说明(Description) |
|---|---|---|
| comments | String[] | 指定行前缀，用来表明哪些是注释行 |
| encoding | String | 指定使用哪种文本编码 - 默认值为 "ISO-8859-1" |
| lineMapper | LineMapper | 将一个 `String` 转换为相应的 `Object`. |
| linesToSkip | int | 在文件顶部有多少行需要跳过/忽略 |
| recordSeparatorPolicy | RecordSeparatorPolicy | 记录分拆策略, 用于确定行尾, 以及如果在引号之中时，如何处理跨行的内容. |
| resource | Resource | 从哪个资源读取数据. |
| skippedLinesCallback | LineCallbackHandler | 忽略输入文件中某些行时, 会将忽略行的原始内容传递给这个回调接口。 如果 `linesToSkip` 设置为**2**, 那么这个接口就会被调用**2**次。 |
| strict | boolean | 如果处于严格模式(strict mode), reader 在 ExecutionContext 中执行时，如果输入资源不存在, 则抛出异常. |

**LineMapper**

就如同 **RowMapper** 在底层根据 ResultSet 构造一个 Object 并返回， 平面文件处理过程中也需要将一行 String 转换并构造成 Object :

```
public interface LineMapper<T> {

    T mapLine(String line, int lineNumber) throws Exception;

}
```

基本的约定是, 给定当前行以及和它关联的行号(line number), mapper 应该能够返回一个领域对象。这类似于在 RowMapper 中每一行也有一个 line number 相关联, 正如 ResultSet 中的每一行(Row)都有其绑定的 row number。这允许行号能被绑定到生成的领域对象以方便比较(identity comparison)或者更方便进行日志记录。

但与 RowMapper 不同的是, LineMapper 只能取得原始行的String值, 正如上面所说, 给你的是一个半成品。 这行文本值必须先被解析为 FieldSet, 然后才可以映射为一个对象,如下所述。

**LineTokenizer**

对将每一行输入转换为 FieldSet 这种操作的抽象是很有必要的, 因为可能会有各种平面文件格式需要转换为 FieldSet。在 Spring Batch中, 对应的接口是 **LineTokenizer**:

```
public interface LineTokenizer {

    FieldSet tokenize(String line);

}
```

使用 **LineTokenizer** 的约定是, 给定一行输入内容(理论上 String 可以包含多行内容), 返回一个表示该行的 FieldSet 对象。这个 FieldSet 接着会传递给 **FieldSetMapper**。Spring Batch 包括以下LineTokenizer实现:

- `DelmitedLineTokenizer` 适用于处理使用分隔符(delimiter)来分隔一条数据中各个字段的文件。最常见的分隔符是逗号(comma),但管道或分号也经常使用。
- `FixedLengthTokenizer` 适用于记录中的字段都是“固定宽度(fixed width)”的文件。每种记录类型中，每个字段的宽度必须先定义。
- `PatternMatchingCompositeLineTokenizer` 通过使用正则模式匹配，来决定对特定的某一行应该使用 LineTokenizers 列表中的哪一个来执行字段拆分。

**FieldSetMapper**

**FieldSetMapper** 接口只定义了一个方法， `mapFieldSet` , 这个方法接收一个 FieldSet 对象，并将其内容映射到一个 object 中。 根据作业需要, 这个对象可以是自定义的 DTO , 领域对象, 或者是简单数组。FieldSetMapper 与 LineTokenizer 结合使用以将资源文件中的一行数据转化为所需类型的对象:

```
public interface FieldSetMapper<T> {

    T mapFieldSet(FieldSet fieldSet);

}
```

这和 JdbcTemplate 中的 RowMapper 是一样的道理。

**DefaultLineMapper**

既然读取平面文件的接口已经定义好了,那很明显我们需要执行以下三个步骤:

1. 从文件中读取一行。
2. 将读取的字符串传给 `LineTokenizer#tokenize()` 方法,以获取一个 **FieldSet**。
3. 将解析后的 **FieldSet** 传给 **FieldSetMapper** ， 然后将 `ItemReader#read()` 方法执行的结果返回给调用者。

上面的两个接口代表了两个不同的任务: 将一行文本转换为 FieldSet, 以及把 FieldSet 映射为一个领域对象。 因为 **LineTokenizer** 的输入对应着 **LineMapper** 的输入(一行), 并且 **FieldSetMapper** 的输出对应着 **LineMapper** 的输出, 所以 SpringBatch 提供了一个使用LineTokenizer和FieldSetMapper的默认实现。**DefaultLineMapper** 就是大多数情况下用户所需要的:

```
public class DefaultLineMapper<T> implements LineMapper<T>, InitializingBean {

    private LineTokenizer tokenizer;

    private FieldSetMapper<T> fieldSetMapper;

    public T mapLine(String line, int lineNumber) throws Exception {
        return fieldSetMapper.mapFieldSet(tokenizer.tokenize(line));
    }

    public void setLineTokenizer(LineTokenizer tokenizer) {
        this.tokenizer = tokenizer;
    }

    public void setFieldSetMapper(FieldSetMapper<T> fieldSetMapper) {
        this.fieldSetMapper = fieldSetMapper;
    }
}
```

上面的功能由一个默认实现类来提供,而不是 reader 本身内置的(以前版本的框架这样干), 让用户可以更灵活地控制解析过程,特别是需要访问原始行的时候。

文件分隔符读取简单示例

下面的例子用来说明一个实际的领域情景。这个批处理作业将从如下文件中读取 football player(足球运动员) 信息:

```
ID,lastName,firstName,position,birthYear,debutYear
"AbduKa00,Abdul-Jabbar,Karim,rb,1974,1996",
"AbduRa00,Abdullah,Rabih,rb,1975,1999",
"AberWa00,Abercrombie,Walter,rb,1959,1982",
"AbraDa00,Abramowicz,Danny,wr,1945,1967",
"AdamBo00,Adams,Bob,te,1946,1969",
"AdamCh00,Adams,Charlie,wr,1979,2003"
```

该文件的内容将被映射为领域对象 **Player**:

```
public class Player implements Serializable {

    private String ID;
    private String lastName;
    private String firstName;
    private String position;
    private int birthYear;
    private int debutYear;

    public String toString() {
        return "PLAYER:ID=" + ID + ",Last Name=" + lastName +
            ",First Name=" + firstName + ",Position=" + position +
            ",Birth Year=" + birthYear + ",DebutYear=" +
            debutYear;
    }

    // setters and getters...
}
```

为了将 FieldSet 映射为 Player 对象, 需要定义一个 `FieldSetMapper` , 返回 player 对象:

```
protected static class PlayerFieldSetMapper implements FieldSetMapper<Player> {
    public Player mapFieldSet(FieldSet fieldSet) {
        Player player = new Player();

        player.setID(fieldSet.readString(0));
        player.setLastName(fieldSet.readString(1));
        player.setFirstName(fieldSet.readString(2));
        player.setPosition(fieldSet.readString(3));
```

```
        player.setBirthYear(fieldSet.readInt(4));
        player.setDebutYear(fieldSet.readInt(5));

        return player;
    }
}
```

然后就可以通过正确构建一个 `FlatFileItemReader`，调用 `read` 方法来读取文件:

```
FlatFileItemReader<Player> itemReader = new FlatFileItemReader<Player>();
itemReader.setResource(new FileSystemResource("resources/players.csv"));
//DelimitedLineTokenizer defaults to comma as its delimiter
LineMapper<Player> lineMapper = new DefaultLineMapper<Player>();
lineMapper.setLineTokenizer(new DelimitedLineTokenizer());
lineMapper.setFieldSetMapper(new PlayerFieldSetMapper());
itemReader.setLineMapper(lineMapper);
itemReader.open(new ExecutionContext());
Player player = itemReader.read();
```

每调用一次 `read` 方法,都会读取文件中的一行，并返回一个新的 `Player` 对象。如果到达文件结尾, 则会返回 `null` 。

根据 **Name** 映射 **Fields**

有一个额外的功能, **DelimitedLineTokenizer** 和 **FixedLengthTokenizer** 都支持，在功能上类似于 Jdbc 的 **ResultSet**。字段的名称可以注入到这些 LineTokenizer 实现以提高映射函数的读取能力。首先, 平面文件中所有字段的列名会注入给 tokenizer:

```
tokenizer.setNames(new String[] {"ID", "lastName","firstName","position","birthYear","debutYear"});
```

FieldSetMapper 可以像下面这样使用此信息:

```
public class PlayerMapper implements FieldSetMapper<Player> {
    public Player mapFieldSet(FieldSet fs) {

       if(fs == null){
           return null;
       }

       Player player = new Player();
       player.setID(fs.readString("ID"));
       player.setLastName(fs.readString("lastName"));
       player.setFirstName(fs.readString("firstName"));
       player.setPosition(fs.readString("position"));
       player.setDebutYear(fs.readInt("debutYear"));
       player.setBirthYear(fs.readInt("birthYear"));

       return player;
   }
}
```

将 **FieldSet** 字段映射为 **Domain Object**

很多时候, 创建一个 FieldSetMapper 就跟 JdbcTemplate 里编写 RowMapper 一样繁琐。Spring Batch通过使用JavaBean规范，提供了一个 FieldSetMapper 来自动将字段映射到对应setter的属性域。还是使用足球的例子, **BeanWrapperFieldSetMapper** 的配置如下所示:

```
<bean id="fieldSetMapper"
      class="org.springframework.batch.item.file.mapping.BeanWrapperFieldSetMapper">
    <property name="prototypeBeanName" value="player" />
</bean>
```

```
<bean id="player"
      class="org.springframework.batch.sample.domain.Player"
      scope="prototype" />
```

对于 FieldSet 中的每个条目(entry), mapper都会在Player对象的新实例中查找相应的setter (因此,需要指定 prototype scope), 和 Spring容器 查找 setter匹配属性名是一样的方式。FieldSet 中每个可用的字段都会被映射, 然后返回组装好的 Player 对象, 不需要再手写代码。

**Fixed Length File Formats**

到这一步,我们讨论了带分隔符的文件, 但实际应用中可能只有一半左右是这种文件。还有很多机构使用固定长度形式的平面文件。固定长度文件的示例如下:

```
UK21341EAH4121131.11customer1
UK21341EAH4221232.11customer2
UK21341EAH4321333.11customer3
UK21341EAH4421434.11customer4
UK21341EAH4521535.11customer5
```

虽然看起来像是一个很长的字段,但实际上代表了4个分开的字段:

1. ISIN : 唯一标识符,订购的商品编码 - 占12字符。
2. Quantity : 订购的商品数量 - 占3字符。
3. Price : 商品的价格 - 占5字符。
4. Customer : 订购商品的顾客Id - 占9字符。

配置好 `FixedLengthLineTokenizer` 以后, 每个字段的长度必须用范围(range)的形式指定:

```
<bean id="fixedLengthLineTokenizer"
      class="org.springframework.batch.io.file.transform.FixedLengthTokenizer">
    <property name="names" value="ISIN,Quantity,Price,Customer" />
    <property name="columns" value="1-12, 13-15, 16-20, 21-29" />
</bean>
```

因为 **FixedLengthLineTokenizer** 使用的也是 LineTokenizer 接口, 所以返回值同样是 FieldSet, 和使用分隔符基本上是一样的。 这也就可以使用同样的方式来处理其输出, 例如使用 **BeanWrapperFieldSetMapper**。

> 注意
>
> 要支持上面这种范围式的语法需要使用专门的属性编辑器： `RangeArrayPropertyEditor` , 可以在 ApplicationContext 中配置。当然,这个 bean 在批处理命名空间中的 ApplicationContext 里已经自动声明了。

单文件中含有多种类型数据的处理

前面所有的文件读取示例，为简单起见都做了一个关键性假设: 在同一个文件中的所有记录都具有相同的格式。但情况有时候并非如此。其实在一个文件包含不同的格式的记录是很常见的，需要使用不同的拆分方式，映射到不同的对象中。下面是一个文件中的片段，仅作演示:

```
USER;Smith;Peter;;T;20014539;F
LINEA;1044391041ABC037.49G201XX1383.12H
LINEB;2134776319DEF422.99M005LI
```

这个文件中有三种类型的记录, "USER", "LINEA", 以及 "LINEB"。 一行 "USER" 对应一个 User 对象。 "LINEA" 和 "LINEB" 对应的都是 Line 对象, 只是 "LINEA" 包含的信息比“LINEB”要多。

ItemReader 分别读取每一行, 当然我们必须指定不同的 LineTokenizer 和 FieldSetMapper 以便ItemWriter 能获得到正确的 item。 **PatternMatchingCompositeLineMapper** 就是专门拿来干这个事的， 可以通过模式映射到对应的 `LineTokenizer` 和 `FieldSetMapper` :

```
<bean id="orderFileLineMapper"
      class="org.spr...PatternMatchingCompositeLineMapper">
    <property name="tokenizers">
        <map>
            <entry key="USER*" value-ref="userTokenizer" />
            <entry key="LINEA*" value-ref="lineATokenizer" />
            <entry key="LINEB*" value-ref="lineBTokenizer" />
        </map>
    </property>
    <property name="fieldSetMappers">
        <map>
            <entry key="USER*" value-ref="userFieldSetMapper" />
            <entry key="LINE*" value-ref="lineFieldSetMapper" />
        </map>
    </property>
</bean>
```

在这个示例中, "LINEA" 和 "LINEB" 使用独立的 LineTokenizer，但使用同一个 FieldSetMapper.

**PatternMatchingCompositeLineMapper** 使用 `PatternMatcher` 的 `match` 方法来为每一行选择正确的代理(delegate)。PatternMatcher 支持两个有特殊的意义通配符(wildcard): 问号(“`?`”， question mark) 将匹配 **1** 个字符(注意不是0-1次), 而星号(“`*`”，asterisk)将匹配 **0** 到多个 字符。

请注意,在上面的配置中,所有以星号结尾的 pattern , 使他们变成了行的有效前缀。 **PatternMatcher** 总是匹配最具体的可能模式, 而不是按配置的顺序从上往下来。所以如果 "`LINE*`" 和 "`LINEA*`" 都配置为 pattern, 那么 "`LINEA`" 将会匹配到 "`LINEA*`", 而 "`LINEB`" 将匹配到 "`LINE*`"。此外,单个星号(“`*`”)可以作为默认匹配所有行的模式，如果该行不匹配其他任何模式的话。

```
<entry key="*" value-ref="defaultLineTokenizer" />
```

还有一个 PatternMatchingCompositeLineTokenizer 可用来单独解析。

在平面文件中，也常常有单条记录跨越多行的情况。要处理这种情况,就需要一种更复杂的策略。这种模式的示例可以参考 第 **11.5** 节 **“**跨域多行的记录**”**。

**Flat File** 的异常处理

在解析一行时, 可能有很多情况会导致异常被抛出。很多平面文件不是很完整, 或者里面的某些记录格式不正确。许多用户会选择忽略这些错误的行, 只将这个问题记录到日志, 比如原始行,行号。稍后可以人工审查这些日志,也可以由另一个批处理作业来检查。出于这个原因,Spring Batch提供了一系列的异常类: `FlatFileParseException` ， 和 `FlatFileFormatException` 。

FlatFileParseException 是由 FlatFileItemReader 在读取文件时解析错误而抛出的。 `FlatFileFormatException` 是由实现了 LineTokenizer 接口的类抛出的, 表明在拆分字段时发生了一个更具体的错误。

**IncorrectTokenCountException**

DelimitedLineTokenizer 和 FixedLengthLineTokenizer 都可以指定列名(column name), 用来创建一个 FieldSet。但如果 column name 的数量和 拆分时找到的列数目， 则不会创建 FieldSet，只会抛出 IncorrectTokenCountException 异常, 里面包含了 字段的实际数量,还有预期的数量:

```
tokenizer.setNames(new String[] {"A", "B", "C", "D"});

try {
```

```
    tokenizer.tokenize("a,b,c");
}
catch(IncorrectTokenCountException e){
    assertEquals(4, e.getExpectedCount());
    assertEquals(3, e.getActualCount());
}
```

因为 tokenizer 配置了4列的名称,但在这个文件中只找到 3 个字段, 所以会抛出 **IncorrectTokenCountException** 异常。

**IncorrectLineLengthException**

固定长度格式的文件在解析时有额外的要求, 因为每一列都必须严格遵守其预定义的宽度。如果一行的总长度不等于所有字段宽度之和, 就会抛出一个异常:

```
tokenizer.setColumns(new Range[] { new Range(1, 5),
                                   new Range(6, 10),
                                   new Range(11, 15) });
try {
    tokenizer.tokenize("12345");
    fail("Expected IncorrectLineLengthException");
}
catch (IncorrectLineLengthException ex) {
    assertEquals(15, ex.getExpectedLength());
    assertEquals(5, ex.getActualLength());
}
```

上面配置的范围是: `1-5` , `6-10` , 以及 `11-15` , 因此预期的总长度是15。但在这里传入的行的长度是 `5` ,所以会导致 IncorrectLineLengthException 异常。之所以直接抛出异常, 而不是先去映射第一个字段的原因是为了更早发现处理失败, 而不再调用 FieldSetMapper 来读取第2列。但是呢, 有些情况下, 行的长度并不总是固定的。 出于这个原因, 可以通过设置 'strict' 属性的值, 不验证行的宽度:

```
tokenizer.setColumns(new Range[] { new Range(1, 5), new Range(6, 10) });
tokenizer.setStrict(false);
FieldSet tokens = tokenizer.tokenize("12345");
assertEquals("12345", tokens.readString(0));
assertEquals("", tokens.readString(1));
```

上面示例和前一个几乎完全相同, 只是调用了 `tokenizer.setStrict(false)` 。这个设置告诉 tokenizer 在对一行进行解析(tokenizing)时不要去管(enforce)行的长度。然后就正确地创建了一个 FieldSet并返回。当然,剩下的值就只会包含空的token值。

# 6.6.3 FlatFileItemWriter

将数据写入到纯文本文件也必须解决和读取文件时一样的问题。 在事务中,一个 step 必须通过分隔符或采用固定长度的格式将数据写出去.

**LineAggregator**

与 `LineTokenizer` 接口的处理方式类似, 写入文件时也需要有某种方式将一条记录的多个字段组织拼接成单个 String,然后再将string写入文件. Spring Batch 对应的接口是 `LineAggregator`:

```
public interface LineAggregator<T> {

    public String aggregate(T item);

}
```

接口 LineAggregator 与 LineTokenizer 相互对应. LineTokenizer 接收 String ,处理后返回一个 FieldSet 对象, 而 LineAggregator 则是接收一条记录,返回对应的 String.

**PassThroughLineAggregator**

`LineAggregator` 接口最基础的实现类是 `PassThroughLineAggregator`, 这个简单实现仅仅是将接收到的对象调用 toString() 方法的值返回:

```
public class PassThroughLineAggregator<T> implements LineAggregator<T> {

    public String aggregate(T item) {
        return item.toString();
    }
}
```

上面的实现对于需要直接转换为string的时候是很管用的,但是 FlatFileItemWriter 的一些优势也是很有必要的,比如 事务,以及支持重启特性等.

简单的文件写入示例

既然已经有了 LineAggregator 接口以及其最基础的实现, PassThroughLineAggregator, 那就可以解释基础的写出流程了:

1. 将要写出的对象传递给 LineAggregator 以获取一个字符串(String).
2. 将返回的 String 写入配置指定的文件中.

下面是 FlatFileItemWriter 中对应的代码:

```
public void write(T item) throws Exception {
    write(lineAggregator.aggregate(item) + LINE_SEPARATOR);
}
```

简单的配置如下所示:

```
<bean id="itemWriter" class="org.spr...FlatFileItemWriter">
    <property name="resource" value="file:target/test-outputs/output.txt" />
    <property name="lineAggregator">
        <bean class="org.spr...PassThroughLineAggregator"/>
    </property>
```

```
</bean>
```

属性提取器 **FieldExtractor**

上面的示例可以应对最基本的文件写入情景。但使用 `FlatFileItemWriter` 时可能更多地是需要将某个领域对象写到文件,因此必须转换到单行之中。 在读取文件时,有以下步骤:

1. 从文件中读取一行.
2. 将这一行字符串传递给 **LineTokenizer#tokenize()** 方法, 以获取 FieldSet 对象
3. 将分词器返回的 FieldSet 传给一个 FieldSetMapper 映射器, 然后将 **ItemReader#read()** 方法得到的结果 return。

文件的写入也很类似, 但步骤正好相反:

1. 将要写入的对象传递给 writer
2. 将领域对象的属性域转换为数组
3. 将结果数组合并(aggregate)为一行字符串

因为框架没办法知道需要将领域对象的哪些字段写入到文件中，所以就需要有一个 FieldExtractor 来将对象转换为数组:

```
public interface FieldExtractor<T> {

    Object[] extract(T item);

}
```

FieldExtractor 的实现类应该根据传入对象的属性创建一个数组， 稍后使用分隔符将各个元素写入文件，或者作为 field-width line 的一部分.

**PassThroughFieldExtractor**

在很多时候需要将一个集合(如 array、Collection, FieldSet等)写出到文件。 从集合中“提取”一个数组那真的是非常简单: 直接进行简单转换即可。 因此在这种场合 PassThroughFieldExtractor 就派上用场了。应该注意,如果传入的对象不是集合类型的, 那么 PassThroughFieldExtractor 将返回一个数组, 其中只包含提取的单个对象。

**BeanWrapperFieldExtractor**

与文件读取一节中所描述的 `BeanWrapperFieldSetMapper` 一样, 通常使用配置来指定如何将领域对象转换为一个对象数组是比较好的办法, 而不用自己写个方法来进行转换。BeanWrapperFieldExtractor 就提供了这类功能:

```
BeanWrapperFieldExtractor<Name> extractor = new BeanWrapperFieldExtractor<Name>();
extractor.setNames(new String[] { "first", "last", "born" });

String first = "Alan";
String last = "Turing";
int born = 1912;

Name n = new Name(first, last, born);
Object[] values = extractor.extract(n);

assertEquals(first, values[0]);
assertEquals(last, values[1]);
assertEquals(born, values[2]);
```

这个 extractor 实现只有一个必需的属性,就是 `names` , 里面用来存放要映射字段的名字。 就像 `BeanWrapperFieldSetMapper` 需要字段名称来将 FieldSet 中的 field 映射到对象的 setter 方法一样, `BeanWrapperFieldExtractor` 需要 names 映射 getter 方法来创建一个对象数组。值得注意的是, names的顺序决定了field在数组中的顺序。

分隔符文件**(Delimited File)**写入示例

最基础的平面文件格式是将所有字段用分隔符(delimiter)来进行分隔(separated)。这可以通过 **DelimitedLineAggregator** 来完成。下面的例子把一个表示客户信用额度的领域对象写出:

```
public class CustomerCredit {

    private int id;
    private String name;
    private BigDecimal credit;

    //getters and setters removed for clarity
}
```

因为使用到了领域对象,所以必须提供 FieldExtractor 接口的实现，当然也少不了要使用的分隔符:

```
<bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="lineAggregator">
        <bean class="org.spr...DelimitedLineAggregator">
            <property name="delimiter" value=","/>
            <property name="fieldExtractor">
                <bean class="org.spr...BeanWrapperFieldExtractor">
                    <property name="names" value="name,credit"/>
                </bean>
            </property>
        </bean>
    </property>
</bean>
```

在这种情况下, 本章前面提到过的 BeanWrapperFieldExtractor 被用来将 CustomerCredit 中的 name 和 credit 字段转换为一个对象数组, 然后在各个字段之间用逗号分隔写入文件。

固定宽度的**(Fixed Width)**文件写入示例

平面文件的格式并不是只有采用分隔符这种类型。许多人喜欢对每个字段设置一定的宽度，这样就能区分各个字段了,这种做法通常被称为“固定宽度, fixed width”。 Spring Batch 通过 FormatterLineAggregator 支持这种文件的写入。使用上面描述的 CustomerCredit 领域对象, 则可以对它进行如下配置:

```
<bean id="itemWriter" class="org.springframework.batch.item.file.FlatFileItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="lineAggregator">
        <bean class="org.spr...FormatterLineAggregator">
            <property name="fieldExtractor">
                <bean class="org.spr...BeanWrapperFieldExtractor">
                    <property name="names" value="name,credit" />
                </bean>
            </property>
            <property name="format" value="%-9s%-2.0f" />
        </bean>
    </property>
</bean>
```

上面的示例大部分看起来是一样的, 只有 format 属性的值不同:

```
<property name="format" value="%-9s%-2.0f" />
```

底层实现采用 Java 5 提供的 `Formatter` 。Java的 Formatter (格式化) 基于C语言的 `printf` 函数功能。关于如何配置 formatter 请参考 Formatter 的javadoc.

### 处理文件创建(Handling File Creation)

FlatFileItemReader 与文件资源的关系很简单。在初始化 reader 时,如果文件存在则打开, 如果文件不存在那就抛出一个异常(exception)。

但是文件的写入就没那么简单了。乍一看可能会觉得跟 FlatFileItemWriter 一样简单直接粗暴: 如果文件存在则抛出异常, 如果不存在则创建文件并开始写入。

但是, 作业的重启有可能会有BUG。 在正常的重启情景中, 约定与前面所想的恰恰相反: 如果文件存在, 则从已知的最后一个正确位置开始写入, 如果不存在, 则抛出异常。

如果此作业(Job)的文件名每次都是一样的那怎么办? 这时候可能需要删除已存在的文件(重启则不删除)。 因为有这些可能性, `FlatFileItemWriter` 有一个属性 `shouldDeleteIfExists`。将这个属性设置为 `true` , 打开 writer 时会将已有的同名文件删除。

# 6.7 XML Item Readers and Writers

Spring Batch为读取XML映射为Java对象以及将Java对象写为XML记录提供了事务基础。

**[注意]XML**流的限制 StAX API 被用在其他XML解析引擎不适合批处理请求 I/O 时的情况(DOM方式把整个输入文件加载到内存中, 而SAX方式在解析过程中需要用户提供回调)。

让我们仔细看看在Spring Batch中 XML输入和输出是如何运行的。 首先,有一些不同于文件读取和写入的概念,但在Spring Batch XML处理中是很常见的。在处理XML时, 并不像读取文本文件(FieldSets)时采取分隔符标记逐行读取的方式, 而是假定XML资源是对应于单条记录的文档片段（'`fragments`'）的集合:

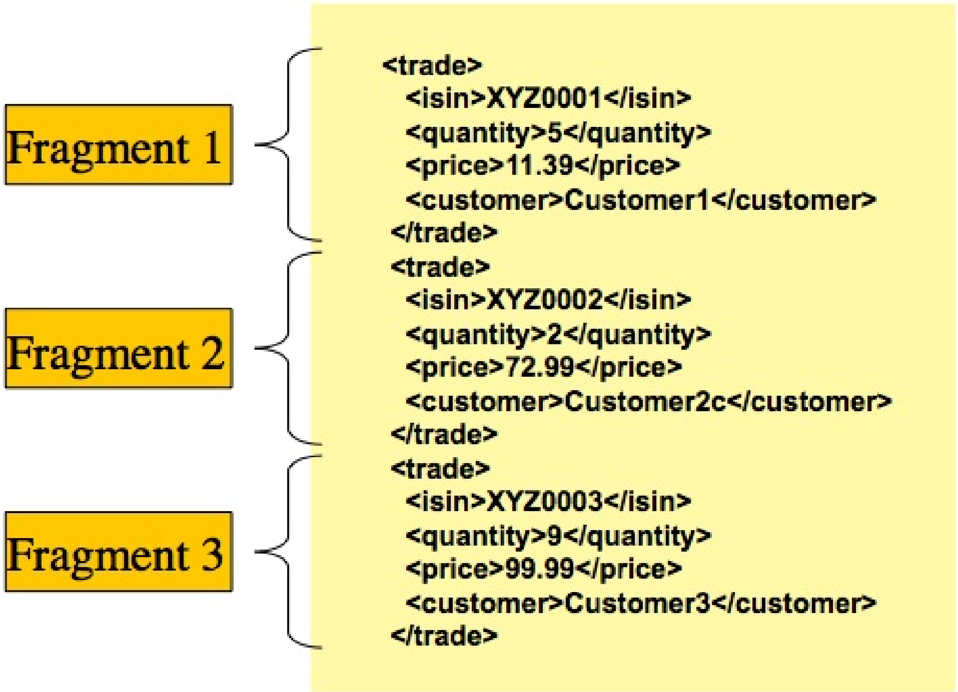



图 **3.1: XML** 输入文件

“`trade`”标签在上面的场景中是根元素“root element”。 在'`<trade>`'和'`</trade>`'之间的一切都被认为是一个 文档片段'`fragment`'。 Spring Batch使用 Object/XML映射(OXM)将 fragments 绑定到对象。 但 Spring Batch 并不依赖某个特定的XML绑定技术。 Spring OXM 委托是最典型的用途, 其为常见的OXM技术提供了统一的抽象。 Spring OXM 依赖是可选的, 如有必要，你也可以自己实现 Spring Batch 的某些接口。 OXM支持的技术间的关系如下图所示:

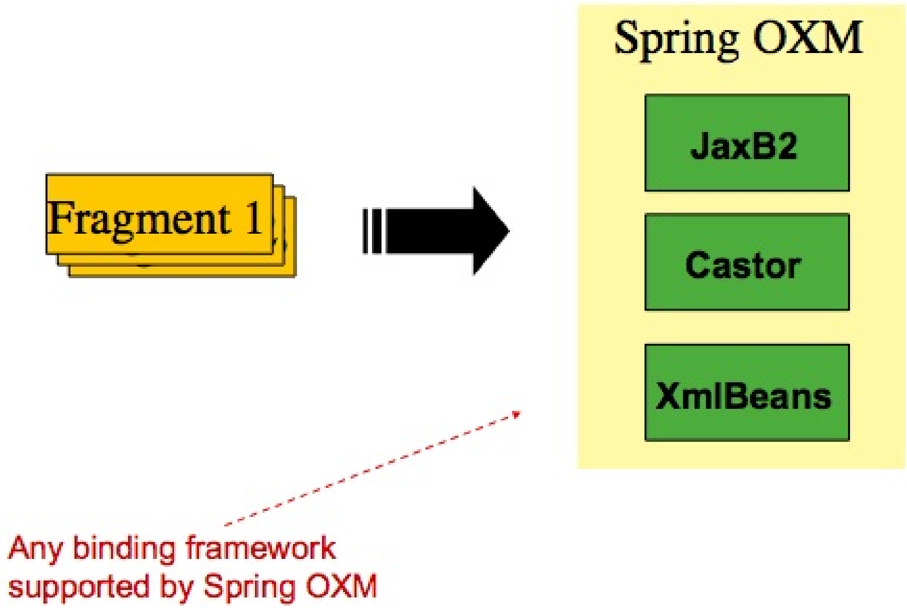



图 **3.2: OXM Binding**

上面介绍了OXM以及如何使用XML片段来表示记录,接着让我们仔细了解下 readers 和 writers 。

# 6.7.1 StaxEventItemReader

**StaxEventItemReader** 提供了从XML输入流进行记录处理的典型设置。 首先,我们来看一下 **StaxEventItemReader**能处理的一组XML记录。

```
<?xml version="1.0" encoding="UTF-8"?>
<records>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0001</isin>
        <quantity>5</quantity>
        <price>11.39</price>
        <customer>Customer1</customer>
    </trade>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0002</isin>
        <quantity>2</quantity>
        <price>72.99</price>
        <customer>Customer2c</customer>
    </trade>
    <trade xmlns="http://springframework.org/batch/sample/io/oxm/domain">
        <isin>XYZ0003</isin>
        <quantity>9</quantity>
        <price>99.99</price>
        <customer>Customer3</customer>
    </trade>
</records>
```

能被处理的XML记录需要满足下列条件:

- **Root Element Name** 片段根元素的名称就是要映射的对象。上面的示例代表的是 trade 的值。
- **Resource** Spring Resource 代表了需要读取的文件。
- **Unmarshaller** Spring OXM提供的Unmarshalling 用于将 XML片段映射为对象.

# #

```
<bean id="itemReader" class="org.springframework.batch.item.xml.StaxEventItemReader">
  <property name="fragmentRootElementName" value="trade" />
  <property name="resource" value="data/iosample/input/input.xml" />
  <property name="unmarshaller" ref="tradeMarshaller" />
</bean>
```

请注意,在上面的例子中,我们选用一个 XStreamMarshaller, 里面接受一个id为 `aliases` 的 map, 将首个entry的 `key` 值作为文档片段的name(即根元素), 将 `value` 作为绑定的对象类型。类似于FieldSet, 后面的其他元素映射为对象内部的字段名/值对。在配置文件中,我们可以像下面这样使用Spring配置工具来描述所需的alias:

```
<bean id="tradeMarshaller"
      class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
        <util:map id="aliases">
            <entry key="trade"
                   value="org.springframework.batch.sample.domain.Trade" />
            <entry key="price" value="java.math.BigDecimal" />
            <entry key="name" value="java.lang.String" />
        </util:map>
    </property>
</bean>
```

当 reader 读取到XML资源的一个新片段时(匹配默认的标签名称)。reader 根据这个片段构建一个独立的XML(或至少看起来

是这样),并将 document 传给反序列化器(通常是一个Spring OXM Unmarshaller 的包装类)将XML映射为一个Java对象。

总之,这个过程类似于下面的Java代码,其中配置了 Spring的注入功能:

```
StaxEventItemReader xmlStaxEventItemReader = new StaxEventItemReader()
Resource resource = new ByteArrayResource(xmlResource.getBytes())

Map aliases = new HashMap();
aliases.put("trade","org.springframework.batch.sample.domain.Trade");
aliases.put("price","java.math.BigDecimal");
aliases.put("customer","java.lang.String");
Marshaller marshaller = new XStreamMarshaller();
marshaller.setAliases(aliases);
xmlStaxEventItemReader.setUnmarshaller(marshaller);
xmlStaxEventItemReader.setResource(resource);
xmlStaxEventItemReader.setFragmentRootElementName("trade");
xmlStaxEventItemReader.open(new ExecutionContext());

boolean hasNext = true

CustomerCredit credit = null;

while (hasNext) {
    credit = xmlStaxEventItemReader.read();
    if (credit == null) {
        hasNext = false;
    }
    else {
        System.out.println(credit);
    }
}
```

## 6.7.2 StaxEventItemWriter

输出与输入相对应. **StaxEventItemWriter** 需要 1个 `Resource`, 1个 `marshaller` 以及 1个 `rootTagName`. Java对象传递给marshaller(通常是标准的Spring OXM marshaller), marshaller 使用自定义的事件writer写入Resource, 并过滤由OXM工具为每条 fragment 产生的 StartDocument 和 EndDocument事件。我们用 MarshallingEventWriterSerializer 示例来显示这一点。Spring配置如下所示:

```
<bean id="itemWriter" class="org.springframework.batch.item.xml.StaxEventItemWriter">
    <property name="resource" ref="outputResource" />
    <property name="marshaller" ref="customerCreditMarshaller" />
    <property name="rootTagName" value="customers" />
    <property name="overwriteOutput" value="true" />
</bean>
```

上面配置了3个必需的属性,以及1个可选属性 `overwriteOutput = true`, (本章前面提到过) 用来指定一个已存在的文件是否可以被覆盖。应该注意的是, `writer` 使用的 `marshaller` 和前面讲的 reading 示例中是完全相同的:

```
<bean id="customerCreditMarshaller"
      class="org.springframework.oxm.xstream.XStreamMarshaller">
    <property name="aliases">
        <util:map id="aliases">
            <entry key="customer"
                   value="org.springframework.batch.sample.domain.CustomerCredit" />
            <entry key="credit" value="java.math.BigDecimal" />
            <entry key="name" value="java.lang.String" />
        </util:map>
    </property>
</bean>
```

我们用一段Java代码来总结所讨论的知识点, 并演示如何通过代码手动设置所需的属性:

```
StaxEventItemWriter staxItemWriter = new StaxEventItemWriter()
FileSystemResource resource = new FileSystemResource("data/outputFile.xml")

Map aliases = new HashMap();
aliases.put("customer","org.springframework.batch.sample.domain.CustomerCredit");
aliases.put("credit","java.math.BigDecimal");
aliases.put("name","java.lang.String");
Marshaller marshaller = new XStreamMarshaller();
marshaller.setAliases(aliases);

staxItemWriter.setResource(resource);
staxItemWriter.setMarshaller(marshaller);
staxItemWriter.setRootTagName("trades");
staxItemWriter.setOverwriteOutput(true);

ExecutionContext executionContext = new ExecutionContext();
staxItemWriter.open(executionContext);
CustomerCredit Credit = new CustomerCredit();
trade.setPrice(11.39);
credit.setName("Customer1");
staxItemWriter.write(trade);
```

# 6.8 多个数据输入文件

在单个 **Step** 中处理多个输入文件是很常见的需求。如果这些文件都有相同的格式, 则可以使用 **MultiResourceItemReader** 来进行处理(支持 XML/或 纯文本文件)。 假如某个目录下有如下3个文件:

```
file-1.txt
file-2.txt
ignored.txt
```

`file-1.txt` 和 `file-2.txt` 具有相同的格式, 根据业务需求需要一起处理. 可以通过 **MuliResourceItemReader** 使用 通配符的形式来读取这两个文件:

```
<bean id="multiResourceReader" class="org.spr...MultiResourceItemReader">
    <property name="resources" value="classpath:data/input/file-*.txt" />
    <property name="delegate" ref="flatFileItemReader" />
</bean>
```

`delegate` 引用的是一个简单的 **FlatFileItemReader**。上面的配置将会从两个输入文件中读取数据,处理回滚以及重启场景。应该注意的是,所有 **ItemReader** 在添加额外的输入文件后(如本示例),如果重新启动则可能会导致某些潜在的问题。 官方建议是每个批作业处理独立的目录,一直到成功完成为止。

# 6.9 数据库(Database)

和大部分企业应用一样,数据库也是批处理系统存储数据的核心机制。 但批处理与其他应用的不同之处在于,批处理系统一般都运行于大规模数据集基础上。 如果一条SQL语句返回100万行, 则结果集可能全部存放在内存中m直到所有行全部读完。Spring Batch提供了两种类型的解决方案来处理这个问题: 游标(Cursor) 和 可分页的数据库ItemReaders.

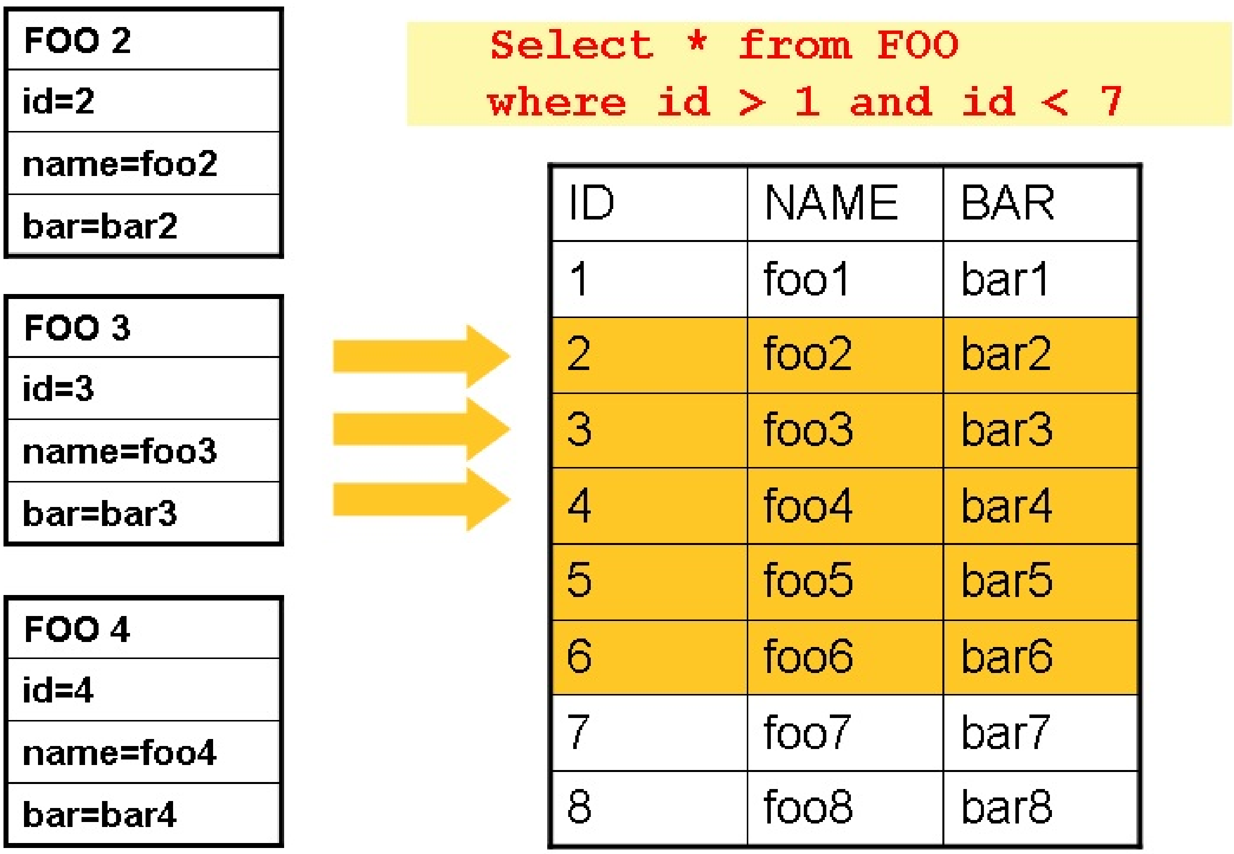
## 6.9.1 基于Cursor的ItemReaders

使用游标(cursor)是大多数批处理开发人员默认采用的方法, 因为它是处理有关系的数据“流”在数据库级别的解决方案。Java 的 `ResultSet` 类其本质就是用面向对象的游标处理机制。 *ResultSet* 维护着一个指向当前数据行的 cursor。调用 *ResultSet* 的 next 方法则将游标移到下一行。

Spring Batch 基于 cursor 的 *ItemReaders* 在初始化时打开游标, 每次调用 read 时则将游标向前移动一行, 返回一个可用于进行处理的映射对象。最好将会调用 close 方法, 以确保所有资源都被释放。

Spring 的 `JdbcTemplate` 的解决办法, 是通过回调模式将 *ResultSet* 中所有行映射之后，在返回调用方法前关闭结果集来处理的。

但是,在批处理的时候就不一样了, 必须得等 step 执行完成才能调用close。下图描绘了基于游标的ItemReader是如何处理的, 使用的SQL语句非常简单, 而且都是类似的实现方式:




这个例子演示了基本的处理模式。 数据库中有一个 “ `FOO` ” 表,它有三个字段: `ID` , `NAME` , 以及 `BAR` , select 查询所有ID大于1但小于7的行。这样的话游标起始于 ID 为 2的行(第1行)。这一行的结果会被映射为一个Foo对象。再次调用read()则将光标移动到下一行, 也就是ID为3的Foo。 在所有行读取完毕之后这些结果将会被写出去, 然后这些对象就会被垃圾回收(假设没有其他引用指向他们)。

> 译注
>
> Foo、Bar 都是英文中的任意代词,没有什么具体意义, 就如我们说的 张三,李四 一样

**JdbcCursorItemReader**

JdbcCursorItemReader 是基于 cursor 的Jdbc实现。它直接使用ResultSet，需要从数据库连接池中获取连接来执行SQL语句。我们的示例使用下面的数据库表:

```
CREATE TABLE CUSTOMER (
   ID BIGINT IDENTITY PRIMARY KEY,
   NAME VARCHAR(45),
   CREDIT FLOAT
);
```

我们一般使用领域对象来对应到每一行, 所以用 `RowMapper` 接口的实现来映射 `CustomerCredit` 对象:

```
public class CustomerCreditRowMapper implements RowMapper {

    public static final String ID_COLUMN = "id";
    public static final String NAME_COLUMN = "name";
    public static final String CREDIT_COLUMN = "credit";

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        CustomerCredit customerCredit = new CustomerCredit();

        customerCredit.setId(rs.getInt(ID_COLUMN));
        customerCredit.setName(rs.getString(NAME_COLUMN));
        customerCredit.setCredit(rs.getBigDecimal(CREDIT_COLUMN));

        return customerCredit;
    }
}
```

一般来说Spring的用户对 `JdbcTemplate` 都不陌生，而 *JdbcCursorItemReader* 使用其作为关键API接口, 我们一起来学习如何通过 `JdbcTemplate` 读取这一数据, 看看它与 `ItemReader` 有何区别。 为了演示方便, 我们假设CUSTOMER表有1000行数据。第一个例子将使用 `JdbcTemplate`:

```
//For simplicity sake, assume a dataSource has already been obtained
JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
List customerCredits = jdbcTemplate.query("SELECT ID, NAME, CREDIT from CUSTOMER",
                                          new CustomerCreditRowMapper());
```

当执行完上面的代码, **customerCredits** 这个 List 中将包含 1000 个 **CustomerCredit** 对象。 在 query 方法中, 先从 **DataSource** 获取一个连接, 然后用来执行给定的SQL, 获取结果后对 **ResultSet** 中的每一行调用一次 `mapRow` 方法。 让我们来对比一下 **JdbcCursorItemReader** 的实现:

```
JdbcCursorItemReader itemReader = new JdbcCursorItemReader();
itemReader.setDataSource(dataSource);
itemReader.setSql("SELECT ID, NAME, CREDIT from CUSTOMER");
itemReader.setRowMapper(new CustomerCreditRowMapper());
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close(executionContext);
```

运行这段代码后 counter 的值将变成 1000。如果上面的代码将返回的 customerCredit 放入 List, 则结果将和使用 **JdbcTemplate** 的例子完全一致。 但是呢, 使用 **ItemReader** 的强大优势在于, 它允许数据项变成 “流式(streamed)”。 调用一次 `read` 方法, 通过**ItemWriter**写出数据对象, 然后再通过 `read` 获取下一项。 这使得 item 读取和写出可以进行 “分块(chunks)”, 并且周期性地提交, 这才是高性能批处理的本质。此外,它可以很容易地通过配置注入到某个 Spring Batch `Step` 中:

```
<bean id="itemReader" class="org.spr...JdbcCursorItemReader">
```

```
    <property name="dataSource" ref="dataSource"/>
    <property name="sql" value="select ID, NAME, CREDIT from CUSTOMER"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
    </property>
</bean>
```

### 附加属性

因为在Java中有很多种不同的方式来打开游标, 所以 `JdbcCustorItemReader` 有许多可以设置的属性：

## 需要整理

**Table 6.2. JdbcCursorItemReader 的属性(Properties)**

| | |
|---|---|
| ignoreWarnings | 决定 SQL警告(SQLWarnings)是否被日志记录,还是导致异常 - 默认值为 true |
| fetchSize | 给 Jdbc driver 一个提示, 当 `ItemReader` 对象需要从 `ResultSet` 中获取更多记录时, 每次应该取多少行数据. 默认没有给定 hint 值. |
| maxRows | 设置底层的 `ResultSet` 最多可以持有多少行记录 |
| queryTimeout | 设置 driver 在执行 `Statement` 对象时应该在给定的时间(单位: 秒)内完成。 如果超过这个时间限制,就抛出一个 `DataAccessEception` 异常.(详细信息请参考/咨询具体数据库驱动的相关文档). |
| verifyCursorPosition | 因为 `ItemReader` 持有的同一个 `ResultSet` 会被传递给 `RowMapper` , 所以用户有可能会自己调用 `ResultSet.next` (), 这就有可能会影响到 reader 内部的计数状态. 将这个值设置为 true 时， 如果在调用 `RowMapper` 前后游标位置(cursor position)不一致,就会抛出一个异常. |
| saveState | 明确指定 reader 的状态是否应该保存在 `ItemStream#update` ( `ExecutionContext` ) 提供的 `ExecutionContext` 中， 默认值为 true. |
| driverSupportsAbsolute | 默认值为 false. 指明 Jdbc 驱动是否支持在 `ResultSet` 中设置绝对行(absolute row). 官方建议,对于支持 `ResultSet.absolute` () 的 Jdbc drivers, 应该设置为 true, 一般能提高效率和性能, 特别是在某个 step 处理很大的数据集失败时. |
| setUseSharedExtendedConnection | 默认值为 false. 指明此 cursor 使用的数据库连接是否和其他处理过程共享连接, 以便处于同一个事务中. 如果设置为 false, 也就是默认值, 那么游标会打开自己的数据库连接, 也就不会参与到 step 处理中的其他事务. 如果要将标志位设置为 true, 则必须将 `DataSource` 包装在一个 `ExtendedConnectionDataSourceProxy` 中, 以阻止每次提交之后关闭/释放连接. 如果此选项设置为 true ,则打开cursor的语句将会自动带上 'READ_ONLY' 和 'HOLD_CUSORS_OVER_COMMIT' 选项. 这样就允许在 step 处理过程中保持 cursor 跨越多个事务. 要使用这个特性,需要数据库服务器的支持, 以及JDBC 驱动符合 Jdbc 3.0 版本规范. |

**HibernateCursorItemReader**

使用 Spring 的程序员需要作出一个重要的决策, 即是否使用ORM解决方案,这决定了是否使用 **JdbcTemplate** 或 **HibernateTemplate** , Spring Batch开发者也面临同样的选择。HibernateCursorItemReader 是 Hibernate 的游标实现。 其实在批处理中使用 Hibernate 那是相当有争议。这很大程度上是因为 Hibernate 最初就是设计了用来开发在线程序的。

但也不是说Hibernate就不能用来进行批处理。最简单的解决办法就是使用一个 StatelessSession (无状态会话), 而不使用标准 session 。这样就去掉了在批处理场景中 Hibernate 那些恼人的缓存、脏检查等等。

更多无状态会话与正常hibernate会话之间的差异, 请参考你使用的 hibernate 版本对应的文档。**HibernateCursorItemReader** 允许您声明一个HQL语句, 并传入 SessionFactory , 然后每次调用 read 时就会返回一个对象, 和 **JdbcCursorItemReader** 一样。下面的示例配置也使用和 JDBC reader 相同的数据库表:

```
HibernateCursorItemReader itemReader = new HibernateCursorItemReader();
```

```
itemReader.setQueryString("from CustomerCredit");
//For simplicity sake, assume sessionFactory already obtained.
itemReader.setSessionFactory(sessionFactory);
itemReader.setUseStatelessSession(true);
int counter = 0;
ExecutionContext executionContext = new ExecutionContext();
itemReader.open(executionContext);
Object customerCredit = new Object();
while(customerCredit != null){
    customerCredit = itemReader.read();
    counter++;
}
itemReader.close(executionContext);
```

这里配置的 ItemReader 将以完全相同的方式返回CustomerCredit对象，和 JdbcCursorItemReader 没有区别, 如果 hibernate 映射文件正确的话。 `useStatelessSession` 属性的默认值为 `true` ，这里明确设置的目的只是为了引起你的注意,我们可以通过他来进行切换。 还值得注意的是 可以通过 `setFetchSize` 设置底层 cursor 的 `fetchSize` 属性 。与 JdbcCursorItemReader一样,配置很简单:

```
<bean id="itemReader"
      class="org.springframework.batch.item.database.HibernateCursorItemReader">
    <property name="sessionFactory" ref="sessionFactory" />
    <property name="queryString" value="from CustomerCredit" />
</bean>
```

#### StoredProcedureItemReader

有时候使用存储过程来获取游标数据是很有必要的。 **StoredProcedureItemReader** 和 **JdbcCursorItemReader** 其实差不多，只是不再执行一个查询来获取游标，而是执行一个存储过程, 由存储过程返回一个游标。 存储过程有三种返回游标的方式:

1. 作为一个 ResultSet 返回(SQL Server, Sybase, DB2, Derby 以及 MySQL支持)
2. 作为一个 out 参数返回 ref-cursor (Oracle和PostgreSQL使用这种方式)
3. 作为存储函数(stored function)的返回值

下面是一个基本的配置示例, 还是使用上面 “客户信用” 的例子:

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
    </property>
</bean>
```

这个例子依赖于存储过程提供一个 ResultSet 作为返回结果(方式1)。

如果存储过程返回一个ref-cursor(方式2),那么我们就需要提供返回的ref-cursor(out 参数)的位置。下面的示例中,第一个参数是返回的ref-cursor:

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="refCursorPosition" value="1"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
    </property>
</bean>
```

如果存储函数的返回值是一个游标(方式 3), 则需要将 `function` 属性设置为 `true` , 默认为 `false` 。如下面所示:

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="sp_customer_credit"/>
    <property name="function" value="true"/>
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.domain.CustomerCreditRowMapper"/>
    </property>
</bean>
```

在所有情况下,我们都需要定义 **RowMapper** 以及 **DataSource**, 还有存储过程的名字。

如果存储过程/函数需要传入参数, 那么必须声明并通过 parameters 属性来设置值。下面是一个关于 Oracle 的示例, 其中声明了三个参数。 第一个是 out 参数,用来返回 ref-cursor, 第二第三个参数是 in 型参数, 类型都是 **INTEGER** :

```
<bean id="reader" class="o.s.batch.item.database.StoredProcedureItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="procedureName" value="spring.cursor_func"/>
    <property name="parameters">
        <list>
            <bean class="org.springframework.jdbc.core.SqlOutParameter">
                <constructor-arg index="0" value="newid"/>
                <constructor-arg index="1">
                    <util:constant static-field="oracle.jdbc.OracleTypes.CURSOR"/>
                </constructor-arg>
            </bean>
            <bean class="org.springframework.jdbc.core.SqlParameter">
                <constructor-arg index="0" value="amount"/>
                <constructor-arg index="1">
                    <util:constant static-field="java.sql.Types.INTEGER"/>
                </constructor-arg>
            </bean>
            <bean class="org.springframework.jdbc.core.SqlParameter">
                <constructor-arg index="0" value="custid"/>
                <constructor-arg index="1">
                    <util:constant static-field="java.sql.Types.INTEGER"/>
                </constructor-arg>
            </bean>
        </list>
    </property>
    <property name="refCursorPosition" value="1"/>
    <property name="rowMapper" ref="rowMapper"/>
    <property name="preparedStatementSetter" ref="parameterSetter"/>
</bean>
```

除了参数声明, 我们还需要指定一个 `PreparedStatementSetter` 实现来设置参数值。这和上面的 JdbcCursorItemReader 一样。查看全部附加属性请查看 附加属性, StoredProcedureItemReader 的附加属性也一样。

## 6.9.2 可分页的 ItemReader

另一种是使用数据库游标执行多次查询,每次查询只返回一部分结果。 我们将这一部分称为一页(a page)。 分页时每次查询必须指定想要这一页的起始行号和想要返回的行数。

### JdbcPagingItemReader

分页 **ItemReader** 的一个实现是 `JdbcPagingItemReader` 。 `JdbcPagingItemReader` 需要一个 **PagingQueryProvider** 来负责提供获取每一页所需的查询SQL。由于每个数据库都有不同的分页策略, 所以我们需要为各种数据库使用对应的 `PagingQueryProvider` 。 也有自动检测所使用数据库类型的 `SqlPagingQueryProviderFactoryBean` ,会根据数据库类型选用适当的 **PagingQueryProvider** 实现。 这简化了配置,同时也是推荐的最佳实践。

**SqlPagingQueryProviderFactoryBean** 需要指定一个 select 子句以及一个 from 子句(clause). 当然还可以选择提供 where 子句. 这些子句加上所需的排序列 sortKey 被组合成为一个 SQL 语句(statement).

在 reader 被打开以后, 每次调用 read 方法则返回一个 item,和其他的 ItemReader一样. 使用分页是因为可能需要额外的行.

下面是一个类似 'customer credit' 示例的例子,使用上面提到的基于 cursor的ItemReaders:

```
<bean id="itemReader" class="org.spr...JdbcPagingItemReader">
    <property name="dataSource" ref="dataSource"/>
    <property name="queryProvider">
        <bean class="org.spr...SqlPagingQueryProviderFactoryBean">
            <property name="selectClause" value="select id, name, credit"/>
            <property name="fromClause" value="from customer"/>
            <property name="whereClause" value="where status=:status"/>
            <property name="sortKey" value="id"/>
        </bean>
    </property>
    <property name="parameterValues">
        <map>
            <entry key="status" value="NEW"/>
        </map>
    </property>
    <property name="pageSize" value="1000"/>
    <property name="rowMapper" ref="customerMapper"/>
</bean>
```

这里配置的ItemReader将返回CustomerCredit对象, 必须指定使用的RowMapper。 '`pageSize`'属性决定了每次数据库查询返回的实体数量。

'`parameterValues`'属性可用来为查询指定参数映射map。如果在where子句中使用了命名参数,那么这些entry的key应该和命名参数一一对应。如果使用传统的 '**?**' 占位符, 则每个entry的key就应该是占位符的数字编号,和JDBC占位符一样索引都是从1开始。

### JpaPagingItemReader

另一个分页ItemReader的实现是 `JpaPagingItemReader` 。JPA没有 Hibernate 中StatelessSession 之类的概念,所以我们必须使用JPA规范提供的其他功能。因为JPA支持分页,所以在使用JPA来处理分页时这是一种很自然的选择。读取每页后, 实体将会分离而且持久化上下文将会被清除,以允许在页面处理完成后实体会被垃圾回收。

**JpaPagingItemReader** 允许您声明一个JPQL语句,并传入一个 **EntityManagerFactory** 。然后就和其他的 ItemReader 一样,每次调用它的 read 方法都会返回一个 item. 当需要更多实体,则内部就会自动发生分页。下面是一个示例配置,和上面的 JDBC reader一样,都是 'customer credit':

```
<bean id="itemReader" class="org.spr...JpaPagingItemReader">
```

```
    <property name="entityManagerFactory" ref="entityManagerFactory"/>
    <property name="queryString" value="select c from CustomerCredit c"/>
    <property name="pageSize" value="1000"/>
</bean>
```

这里配置的ItemReader和前面所说的 JdbcPagingItemReader 返回一样的 CustomerCredit对象, 假设 Customer 对象有正确的JPA注解或者ORM映射文件。 '`pageSize`' 属性决定了每次查询时读取的实体数量。

## IbatisPagingItemReader

> **[Note]** 注意事项
>
> 这个 reader 在 Spring Batch 3.0中已经被废弃(deprecated).

如果使用 IBATIS/MyBatis, 则可以使用 IbatisPagingItemReader, 顾名思义, 也是一种实现分页的ItemReader。IBATIS不对分页提供直接支持, 但通过提供一些标准变量就可以为IBATIS查询提供分页支持。

下面是和上面的示例同样功能的配置,使用IbatisPagingItemReader来读取CustomerCredits:

```
<bean id="itemReader" class="org.spr...IbatisPagingItemReader">
    <property name="sqlMapClient" ref="sqlMapClient"/>
    <property name="queryId" value="getPagedCustomerCredits"/>
    <property name="pageSize" value="1000"/>
</bean>
```

上述 **IbatisPagingItemReader** 配置引用了一个IBATIS查询,名为“**getPagedCustomerCredits**”。如果使用MySQL,那么查询XML应该类似于下面这样。

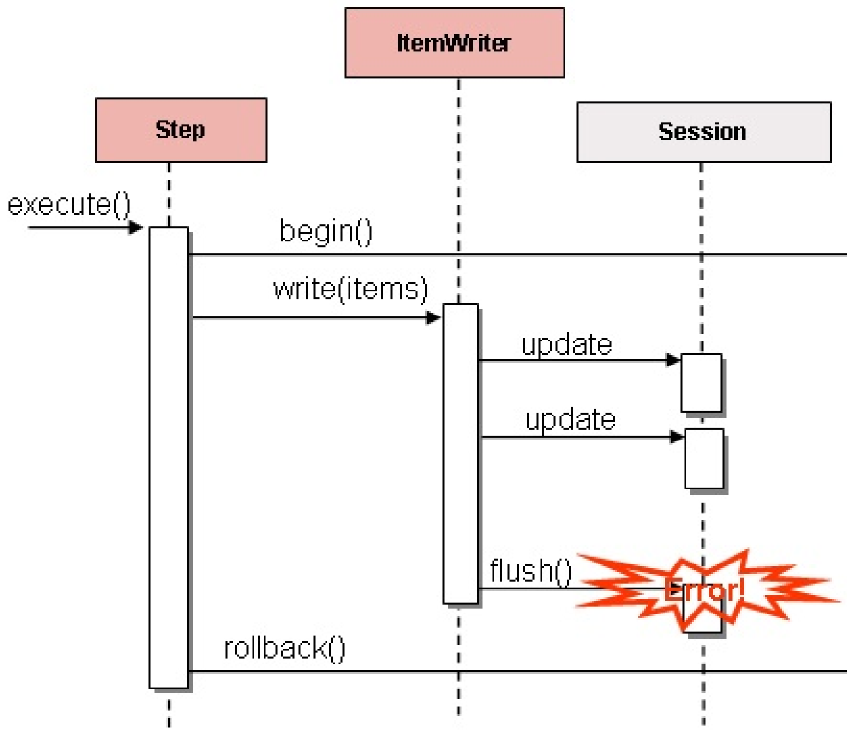
```
<select id="getPagedCustomerCredits" resultMap="customerCreditResult">
    select id, name, credit from customer order by id asc LIMIT #_skiprows#, #_pagesize#
</select>
```

`_skiprows` 和 `_pagesize` 变量都是 IbatisPagingItemReader 提供的,还有一个 `_page` 变量,需要时也可以使用。分页查询的语法根据数据库不同使用。下面是使用Oracle的一个例子(但我们需要使用CDATA来包装某些特殊符号,因为是放在XML文档中嘛):

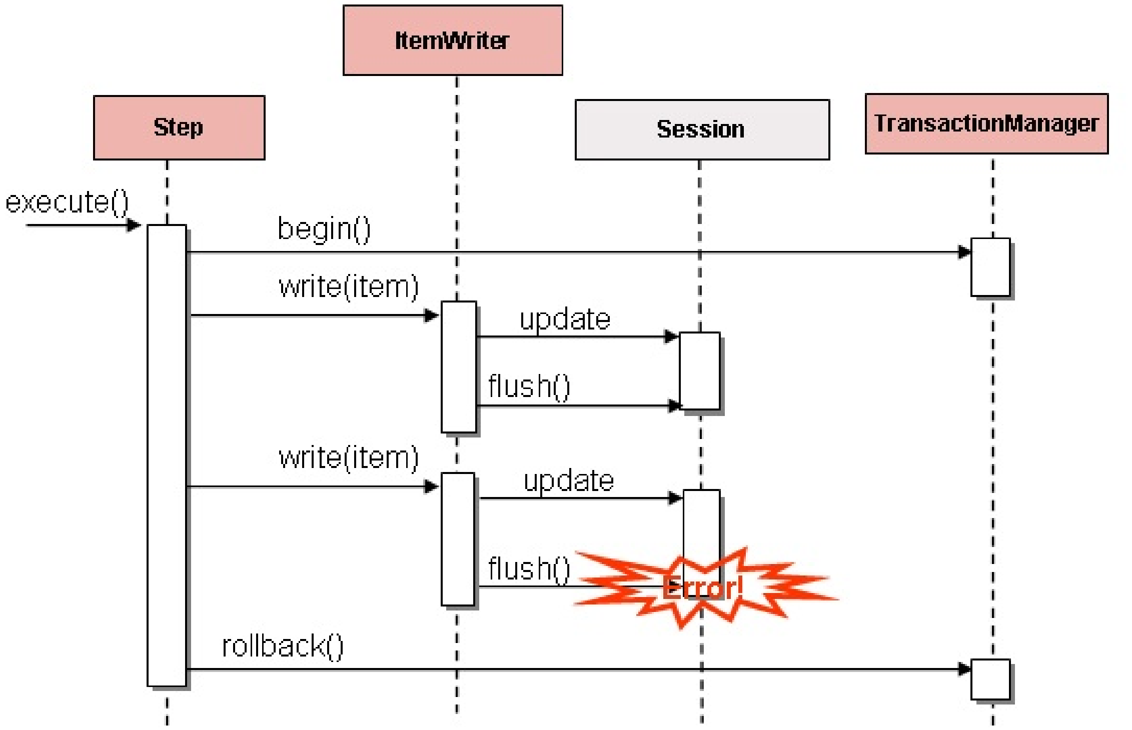
```
<select id="getPagedCustomerCredits" resultMap="customerCreditResult">
    select * from (
      select * from (
        select t.id, t.name, t.credit, ROWNUM ROWNUM_ from customer t order by id
       )) where ROWNUM_ <![CDATA[ > ]]> ( #_page# * #_pagesize# )
    ) where ROWNUM <![CDATA[ <= ]]> #_pagesize#
</select>
```

## 6.9.3 Database ItemWriters

虽然文本文件和XML都有自己特定的 ItemWriter, 但数据库和他们并不一样。这是因为事务提供了所需的全部功能。 对于文件来说 ItemWriters 是必要的, 因为如果需要事务特性,他们必须充当这种角色, 跟踪输出的 item,并在适当的时间 flushing/clearing。使用数据库时不需要这个功能,因为写已经包含在事务之中。 用户可以自己创建实现ItemWriter接口的 DAO, 或使用一个处理常见问题的自定义ItemWriter,无论哪种方式,都不会有任何问题。 需要注意的一件事是批量输出时的性能和错误处理能力。 在使用hibernate作为ItemWriter 时是最常见的, 但在使用Jdbc batch 模式时可能也会存在同样的问题。批处理数据库输出没有任何固有的缺陷,如果我们注意 flush 并且数据没有错误的话。 但是,在写出时如果发生了什么错误,就可能会引起混乱,因为没有办法知道是哪个item引起的异常, 甚至是否某个单独的 item 负有责任,如下图所示:




如果 items 在输出之前有缓冲, 则遇到任何错误将不会立刻抛出, 直到缓冲区刷新之后,提交之前才会抛出。例如, 我们假设每一块写出20个item, 第15个 item 会抛出 `DataIntegrityViolationException` 。如果与 Step 有关, 则20项数据都会写入成功, 因为没有办法知道会出现错误,直到全部写入完成。一旦调用 Session#flush(), 就会清空缓冲区buffer, 而异常也将被放出来。在这一点上, Step无能为力, 事务也必须回滚。 通常, 异常会导致 item 被跳过(取决于 skip/retry 策略), 然后该item就不会被输出。 然而,在批处理的情况下, 是没有办法知道到底是哪一项引起的问题, 在错误发生时整个缓冲区都将被写出。解决这个问题的唯一方法就是在每一个 item之后 flush 一下:




这种用法是很常见的, 尤其是在使用Hibernate时,ItemWriter的简单实现建议, 在每次调用 write() 之后执行 flush。这样做可以让跳过 items 变得可靠, 而Spring Batch 在错误发生后会在内部关注适当粒度的ItemWriter调用。

# 6.10 重用已存在的 Service

批处理系统通常是与其他应用程序相结合的方式使用。最常见的是与一个在线应用系统结合, 但也支持与瘦客户端集成,通过移动每个程序所使用的批量数据。由于这些原因,所以很多用户想要在批处理作业中重用现有的DAO或其他服务。Spring容器通过注入一些必要的类就可以实现这些重用。但可能需要现有的服务作为 **ItemReader** 或者 **ItemWriter**, 也可以适配另一个Spring Batch类, 或其本身就是一个 step 主要的**ItemReader**。为每个需要包装的服务编写一个适配器类是很简单的, 而因为这是很普遍的需求,所以 Spring Batch 提供了实现: `ItemReaderAdapter` 和 `ItemWriterAdapter`。两个类都实现了标准的Spring方法委托模式调用, 设置也相当简单。下面是一个reader的示例:

```
<bean id="itemReader" class="org.springframework.batch.item.adapter.ItemReaderAdapter">
    <property name="targetObject" ref="fooService" />
    <property name="targetMethod" value="generateFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

特别需要注意的是, `targetMethod` 必须和 **read** 方法行为对等: 如果不存在则返回null, 否则返回一个 **Object**。 其他的值会使框架不知道何时该结束处理, 或者引起无限循环或不正确的失败,这取决于 **ItemWriter** 的实现。 **ItemWriter** 的实现同样简单:

```
<bean id="itemWriter" class="org.springframework.batch.item.adapter.ItemWriterAdapter">
    <property name="targetObject" ref="fooService" />
    <property name="targetMethod" value="processFoo" />
</bean>

<bean id="fooService" class="org.springframework.batch.item.sample.FooService" />
```

# 6.11 输入校验

在本章中, 已经讨论了很多种用来解析 input 的方法。 如果格式不对,那这些基本的实现都是抛出异常。 如果数据丢失一部分, **FixedLengthTokenizer** 也会抛出异常。 同样, 使用 **FieldSetMapper** 时,如果读取超出 **RowMapper** 索引范围的值,又或者返回值类型不匹配,都会抛出异常。 所有的异常都会在 **read** 返回之前抛出。 然而, 他们不能确定返回的item是否是合法的。 例如, 如果其中一个字段是 `age` , 很显然不能是负数。 解析为数字是没问题的, 因为确实存在这个数, 所以就不会抛出异常。 因为当下已经有大量的第三方验证框架, 所以 Spring Batch 并不提供另一个验证框架, 而是提供了一个非常简单的接口, 其他框架可以实现这个接口来提供兼容:

```
public interface Validator {

    void validate(Object value) throws ValidationException;
}
```

The contract is that the **validate** method will throw an exception if the object is invalid, and return normally if it is valid. Spring Batch provides an out of the box **ItemProcessor**:

约定是如果对象无效则 `validate` 方法抛出一个异常, 如果对象合法那就正常返回。 Spring Batch 提供了开箱即用的 **ItemProcessor**:

```
<bean class="org.springframework.batch.item.validator.ValidatingItemProcessor">
    <property name="validator" ref="validator" />
</bean>

<bean id="validator"
      class="org.springframework.batch.item.validator.SpringValidator">
    <property name="validator">
        <bean id="orderValidator"
              class="org.springmodules.validation.valang.ValangValidator">
            <property name="valang">
                <value>
                    <![CDATA[
           { orderId : ? > 0 AND ? <= 9999999999 : 'Incorrect order ID' : 'error.order.id' }
           { totalLines : ? = size(lineItems) : 'Bad count of order lines'
                                              : 'error.order.lines.badcount'}
           { customer.registered : customer.businessCustomer = FALSE OR ? = TRUE
                                 : 'Business customer must be registered'
                                 : 'error.customer.registration'}
           { customer.companyName : customer.businessCustomer = FALSE OR ? HAS TEXT
                                  : 'Company name for business customer is mandatory'
                                  :'error.customer.companyname'}
                    ]]>
                </value>
            </property>
        </bean>
    </property>
</bean>
```

这个示例展示了一个简单的 **ValangValidator**, 用来校验 `order` 对象。 这样写目的是为了尽可能多地演示如何使用 Valang 来添加校验程序。

# 6.12 不保存执行状态

默认情况下,所有 **ItemReader** 和 **ItemWriter** 在提交之前都会把当前状态信息保存到 **ExecutionContext** 中。 但有时我们又不希望保存这些信息。 例如,许多开发者使用处理指示器(process indicator)让数据库读取程序 '可重复运行(rerunnable)'。 在数据表中添加一个附加列来标识该记录是否已被处理。 当某条记录被读取/写入时,就将标志位从 `false` 变为 `true` , 然后只要在SQL语句的where子句中包含一个附加条件, 如 "`where PROCESSED_IND = false`", 就可确保在任务重启后只查询到未处理过的记录。 这种情况下,就不需要保存任何状态信息, 比如当前 row number 什么的, 因为在重启后这些信息都没用了。 基于这种考虑, 所有的 readers 和 writers 都含有一个 `saveState` 属性:

```
<bean id="playerSummarizationSource" class="org.spr...JdbcCursorItemReader">
    <property name="dataSource" ref="dataSource" />
    <property name="rowMapper">
        <bean class="org.springframework.batch.sample.PlayerSummaryMapper" />
    </property>
    <property name="saveState" value="false" />
    <property name="sql">
        <value>
            SELECT games.player_id, games.year_no, SUM(COMPLETES),
            SUM(ATTEMPTS), SUM(PASSING_YARDS), SUM(PASSING_TD),
            SUM(INTERCEPTIONS), SUM(RUSHES), SUM(RUSH_YARDS),
            SUM(RECEPTIONS), SUM(RECEPTIONS_YARDS), SUM(TOTAL_TD)
            from games, players where players.player_id =
            games.player_id group by games.player_id, games.year_no
        </value>
    </property>
</bean>
```

上面配置的这个 **ItemReader** 在任何情况下都不会将 entries（状态信息）存放到 **ExecutionContext** 中.

# 6.13 创建自定义 ItemReaders 与 ItemWriters

到目前为止,本章已将 Spring Batch 中基本的读取(reading)和写入(writing)概念讲完, 还对一些常用的实现进行了讨论。然而, 这些都是相当普通的, 还有很多潜在的场景可能没有现成的实现。本节将通过一个简单的例子,来演示如何创建自定义的 `ItemReader` 和 `ItemWriter`,并且如何正确地实现和使用。 `ItemReader` 同时也将 `ItemStream`, 以说明如何让reader(读取器)或writer(写入器)支持重启(restartable)。

## 6.13.1 自定义 ItemReader 示例

为了实现这个目的,我们实现一个简单的 `ItemReader`, 从给定的list中读取数据。 我们将实现最基本的 *ItemReader* 功能, read:

```
public class CustomItemReader<T> implements ItemReader<T>{

    List<T> items;

    public CustomItemReader(List<T> items) {
        this.items = items;
    }

    public T read() throws Exception, UnexpectedInputException,
       NoWorkFoundException, ParseException {

        if (!items.isEmpty()) {
            return items.remove(0);
        }
        return null;
    }
}
```

这是一个简单的类, 传入一个 items list, 每次读取时删除其中的一条并返回。 如果list里面没有内容,则将返回null, 从而满足 `ItemReader` 的基本要求, 测试代码如下所示:

```
List<String> items = new ArrayList<String>();
items.add("1");
items.add("2");
items.add("3");

ItemReader itemReader = new CustomItemReader<String>(items);
assertEquals("1", itemReader.read());
assertEquals("2", itemReader.read());
assertEquals("3", itemReader.read());
assertNull(itemReader.read());
```

使 **`ItemReader`** 支持重启

现在剩下的问题就是让 *ItemReader* 变为可重启的。到目前这一步,如果发生掉电之类的情况,那么必须重新启动 *ItemReader*, 而且是从头开始。在很多时候这是允许的,但有时候更好的处理办法是让批处理作业在上次中断的地方重新开始。判断的关键是根据 reader 是有状态的还是无状态的。 无状态的 reader 不需要考虑重启的情况, 但有状态的则需要根据其最后一个已知的状态来重新启动。出于这些原因, 官方建议尽可能地让 reader 成为无状态的,使开发者不需要考虑重新启动的情况。

如果需要保存状态信息,那应该使用 `ItemStream` 接口:

```
public class CustomItemReader<T> implements ItemReader<T>, ItemStream {

    List<T> items;
    int currentIndex = 0;
    private static final String CURRENT_INDEX = "current.index";

    public CustomItemReader(List<T> items) {
        this.items = items;
    }

    public T read() throws Exception, UnexpectedInputException,
        ParseException {

        if (currentIndex < items.size()) {
            return items.get(currentIndex++);
        }
```

```
        return null;
    }

    public void open(ExecutionContext executionContext) throws ItemStreamException {
        if(executionContext.containsKey(CURRENT_INDEX)){
            currentIndex = new Long(executionContext.getLong(CURRENT_INDEX)).intValue();
        }
        else{
            currentIndex = 0;
        }
    }

    public void update(ExecutionContext executionContext) throws ItemStreamException {
        executionContext.putLong(CURRENT_INDEX, new Long(currentIndex).longValue());
    }

    public void close() throws ItemStreamException {}
}
```

每次调用 *ItemStream* 的 `update` 方法时, *ItemReader* 的当前 `index` 都会被保存到给定的 **ExecutionContext** 中,key 为 '`current.index`'。 当调用 *ItemStream* 的 `open` 方法时, **ExecutionContext**会检查是否包含该 key 对应的条目。 如果找到key, 那么当前索引 index 就好移动到该位置。这是一个相当简单的例子,但它仍然符合通用原则:

```
ExecutionContext executionContext = new ExecutionContext();
((ItemStream)itemReader).open(executionContext);
assertEquals("1", itemReader.read());
((ItemStream)itemReader).update(executionContext);

List<String> items = new ArrayList<String>();
items.add("1");
items.add("2");
items.add("3");
itemReader = new CustomItemReader<String>(items);

((ItemStream)itemReader).open(executionContext);
assertEquals("2", itemReader.read());
```

大多数ItemReaders具有更加复杂的重启逻辑。 例如 **JdbcCursorItemReader** , 存储了游标(Cursor)中最后所处理的行的row id。

还值得注意的是 **ExecutionContext** 中使用的 key 不应该过于简单。这是因为 *ExecutionContext* 被一个 `Step` 中的所有 *ItemStreams* 共用。在大多数情况下,使用类名加上 key 的方式应该就足以保证唯一性。然而,在极端情况下, 同一个类的多个 *ItemStream* 被用在同一个Step中时( 如需要输出两个文件的情况),就需要更加具备唯一性的name标识。出于这个原因,Spring Batch 的许多`ItemReader` 和 `ItemWriter` 实现都有一个 `setName()` 方法, 允许覆盖默认的 key name。

## 6.13.2 自定义 ItemWriter 示例

自定义实现 `ItemWriter` 和上一小节所讲的 `ItemReader` 有很多方面是类似, 但也有足够多的不同之处。 但增加可重启特性在本质上是一样的, 所以本节的示例就不再讨论这一点。和 `ItemReader` 示例一样, 为了简单我们使用的参数也是 `List`:

```
public class CustomItemWriter<T> implements ItemWriter<T> {

    List<T> output = TransactionAwareProxyFactory.createTransactionalList();

    public void write(List<? extends T> items) throws Exception {
        output.addAll(items);
    }

    public List<T> getOutput() {
        return output;
    }
}
```

### 让 `ItemWriter` 支持重新启动

要让 `ItemWriter` 支持重新启动，我们将会使用和 `ItemReader` 相同的过程, 实现并添加 `ItemStream` 接口来同步 execution context。 在示例子中我们可能要记录处理过的items数量,并添加为到 footer 记录。 我们可以在 `ItemWriter` 的实现类中同时实现 `ItemStream`, 以便在 stream 重新打开时从执行上下文中取回原来的数据重建计数器。

实际开发中, 如果自定义 ItemWriter restartable(支持重启),则会委托另一个 writer(例如, 在写入文件时), 否则会写入到关系型数据库(支持事务的资源)中, 此时 ItemWriter 不需要 restartable特性,因为自身是无状态的。 如果你的 writer 有状态, 则应该实现2个接口: `ItemStream` 和 `ItemWriter` 。 请记住, writer客户端需要知道 *ItemStream* 的存在, 所以需要在 xml 配置文件中将其注册为 stream.

# 7.扩展与并行处理

很多批处理问题都可以通过单进程、单线程的工作模式来完成, 所以在想要做一个复杂设计和实现之前,请审查你是否真的需要那些超级复杂的实现。 衡量实际作业(job)的性能,看看最简单的实现是否能满足需求: 即便是最普通的硬件,也可以在一分钟内读写上百MB数据文件。

当你准备使用并行处理技术来实现批处理作业时,Spring Batch提供一系列选择,本章将对他们进行讲述,虽然某些功能不在本章中涵盖。从高层次的抽象角度看，并行处理有两种模式: 单进程,多线程模式; 或者多进程模式。还可以将他分成下面这些种类:

- 多线程Step(单个进程)
- 并行Steps(单个进程)
- 远程分块Step(多个进程)
- 对Step分区 (单/多个进程)

下面我们先回顾一下单进程方式,然后再看多进程方式.

# 7.1 多线程 Step

启动并行处理最简单的方式就是在 Step 配置中加上一个**TaskExecutor** , 比如,作为 **tasklet** 的一个属性:

```
<step id="loading">
    <tasklet task-executor="taskExecutor">...</tasklet>
</step>
```

上面的示例中, taskExecutor指向了另一个实现 **TaskExecutor** 接口的Bean. **TaskExecutor** 是一个标准的Spring接口,具体有哪些可用的实现类,请参考 Spring用户指南. 最简单的多线程 **TaskExecutor** 实现是 **SimpleAsyncTaskExecutor**.

以上配置的结果就是在 Step 在(每次提交的块)记录的读取, 处理, 写入时都会在单独的线程中执行。请注意,这段话的意思就是在要处理的数据项之间没有了固定的顺序, 并且一个非连续块可能包含项目相比,单线程的例子。此外executor还有一些限制(例如,如果它是由一个线程池在后台执行的),有一个tasklet的配置项可以调整,throttle-limit默认为4。你可能根据需要增加这个值以确保线程池被充分利用,如:

```
<step id="loading"> <tasklet
    task-executor="taskExecutor"
    throttle-limit="20">...</tasklet>
</step>
```

还需要注意在step中并发使用连接池资源时可能会有一些限制,例如数据库连接池 **DataSource**. 请确保连接池中的资源数量大于或等于并发线程的数量.

在一些常见的批处理情景中,对使用多线程Step有一些实际的限制。Step中的许多部分(如readers 和 writers)是有状态的,如果某些状态没有进行线程隔离,那么这些组件在多线程Step中就是不可用的。特别是大多数Spring Batch提供的readers 和 writers不是为多线程而设计的。但是,我们也可以使用无状态或线程安全的readers 和 writers,可以参考Spring Batch Samples中(parallelJob)的这个示例(点击进入Section 6.12, “Preventing State Persistence”),示例中展示了通过指示器来跟踪数据库input表中的哪些项目已经被处理过,而哪些还没有被处理。

Spring Batch 提供了**ItemWriter** 和 **ItemReader** 的一些实现. 通常在javadoc中会指明是否是线程安全的,或者指出在并发环境中需要注意哪些问题. 假若文档中没有明确说明,你只能通过查看源代码来看看是否有什么线程不安全的共享状态. 一个并非线程安全的 reader , 也可以在你自己处理了同步的代理对象中高效地使用.

如果你的step中写操作和处理操作所消耗的时间更多,那么即使你对**read()**操作加锁进行同步,也会比你在单线程环境中执行要快很多.

# 7.2 并行 Steps

只要需要并行的程序逻辑可以划分为不同的职责,并分配给各个独立的step,那么就可以在单个进程中并行执行。并行Step执行很容易配置和使用,例如,将执行步骤(**step1**,**step2**)和步骤3**step3**并行执行,则可以向下面这样配置一个流程:

```
<job id="job1">
    <split id="split1" task-executor="taskExecutor" next="step4">
        <flow>
            <step id="step1" parent="s1" next="step2"/>
            <step id="step2" parent="s2"/>
        </flow>
        <flow>
            <step id="step3" parent="s3"/>
        </flow>
    </split>
    <step id="step4" parent="s4"/>
</job>

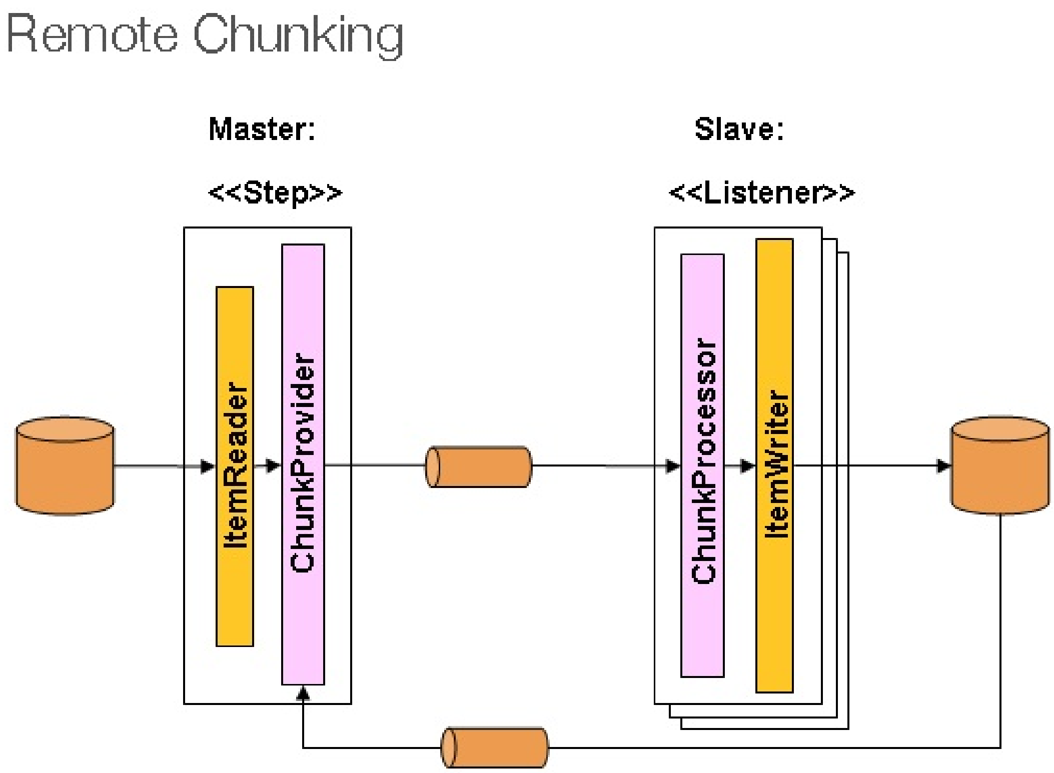
<beans:bean id="taskExecutor" class="org.spr...SimpleAsyncTaskExecutor"/>
```

可配置的 "task-executor" 属性是用来指明应该用哪个TaskExecutor实现来执行独立的流程。默认是**SyncTaskExecutor**,但有时需要使用异步的TaskExecutor来并行运行某些步骤。请注意,这项工作将确保每一个流程在聚合之前完成.并进行过渡。

更详细的信息请参考 Section 5.3.5, “Split Flows”.

# 7.3 远程分块(Remote Chunking)

使用远程分块的Step被拆分成多个进程进行处理,多个进程间通过中间件实现通信. 下面是一幅模型示意图:



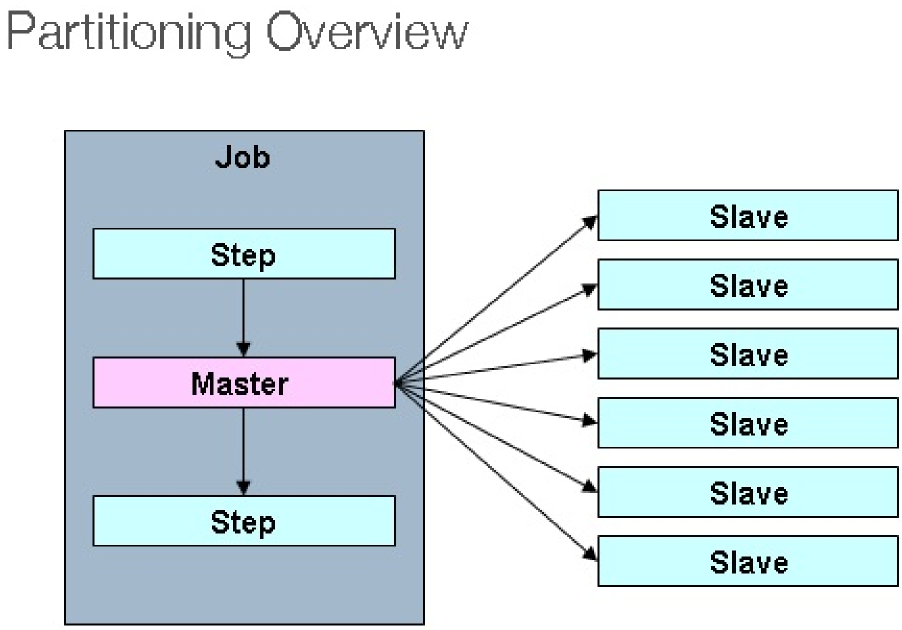

Master组件是单个进程,从属组件(Slaves)一般是多个远程进程。如果Master进程不是瓶颈的话,那么这种模式的效果几乎是最好的,因此应该在处理数据比读取数据消耗更多时间的情况下使用(实际应用中常常是这种情形)。

Master组件只是Spring Batch **Step** 的一个实现, 只是将ItemWriter替换为一个通用的版本,这个通用版本 "知道" 如何将数据项的分块作为消息(messages)发送给中间件。 从属组件(Slaves)是标准的监听器(listeners),不论使用哪种中间件(如使用JMS时就是 **MesssageListeners** ), Slaves的作用都是处理数据项的分块(chunks), 可以使用标准的 **ItemWriter** 或者是 **ItemProcessor**加上一个 **ItemWriter**, 使用的接口是 **ChunkProcessor** interface。使用此模式的一个优点是： reader, processor和 writer 组件都是现成的(就和在本机执行的step一样)。数据项被动态地划分,工作是通过中间件共享的,因此,如果监听器都是饥饿模式的消费者,那么就自动实现了负载平衡。

中间件必须持久可靠,能保证每个消息都会被分发,且只分发给单个消费者。JMS是很受欢迎的解决方案,但在网格计算和共享内存产品空间里还有其他可选的方式(如 Java Spaces服务; 为Java对象提供分布式的共享存储器)。

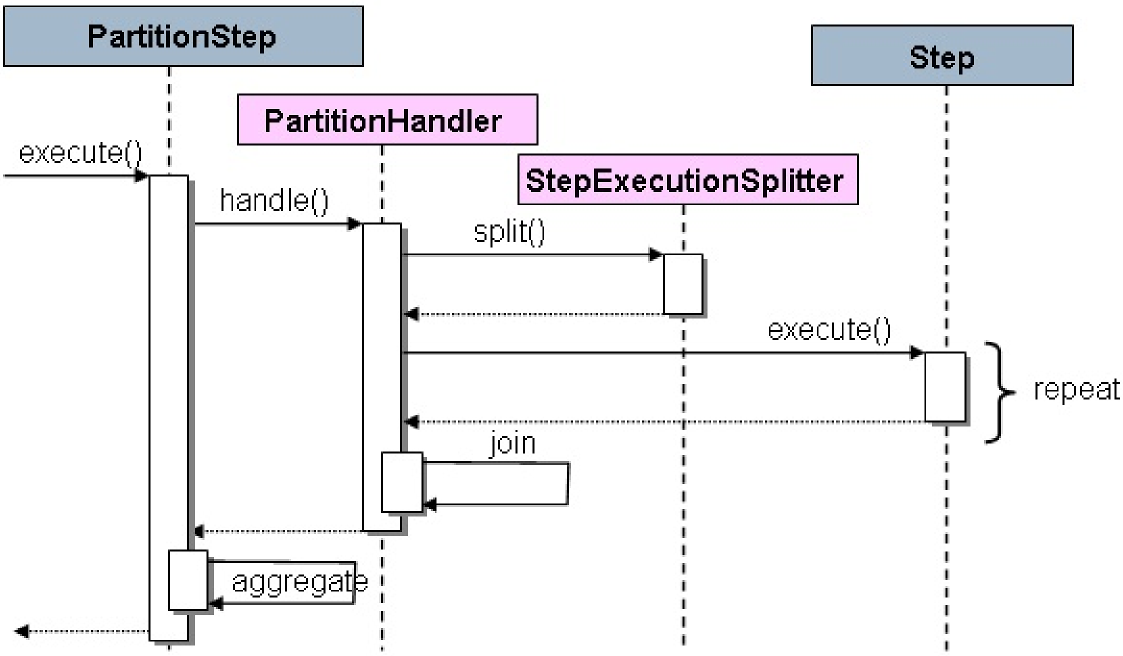
# 7.4 分区

Spring Batch也为Step的分区执行和远程执行提供了一个SPI(服务提供者接口)。在这种情况下,远端的执行程序只是一些简单的Step实例,配置和使用方式都和本机处理一样容易。下面是一幅实际的模型示意图:




在左侧执行的作业(Job)是串行的Steps,而中间的那一个Step被标记为 Master。图中的 Slave 都是一个Step的相同实例,对于作业来说,这些Slave的执行结果实际上等价于就是Master的结果。Slaves通常是远程服务,但也有可能是本地执行的其他线程。在此模式中,Master发送给Slave的消息不需要持久化(durable) ,也不要求保证交付: 对每个作业执行步骤来说,保存在 **JobRepository** 中的Spring Batch元信息将确保每个Slave都会且仅会被执行一次。

Spring Batch的SPI由Step的一个专门的实现( **PartitionStep**),以及需要由特定环境实现的两个策略接口组成。这两个策略接口分别是 **PartitionHandler** 和 **StepExecutionSplitter**,他们的角色如下面的序列图所示:




此时在右边的Step就是“远程”Slave,所以可能会有多个对象 和/或 进程在扮演这一角色,而图中的 PartitionStep 在驱动(/控制)整个执行过程。PartitionStep的配置如下所示:

```
<step id="step1.master">
    <partition step="step1" partitioner="partitioner">
        <handler grid-size="10" task-executor="taskExecutor"/>
    </partition>
</step>
```

类似于多线程step的 throttle-limit 属性, grid-size属性防止单个Step的任务执行器过载。

在Spring Batch Samples示例程序中有一个简单的例子在单元测试中可以拷贝/扩展(详情请参考 ***PartitionJob.xml** 配置文件)。

Spring Batch 为分区创建执行步骤,名如“step1:partition0”,等等,所以我们经常把Master step叫做“step1:master”。在Spring 3.0中也可以为Step指定别名(通过指定 **name** 属性,而不是 **id** 属性)。

## 7.4.1 分区处理器(PartitionHandler)

**PartitionHandler**组件知道远程网格环境的组织结构。 它可以发送**StepExecution**请求给远端Steps,采用某种具体的数据格式,例如DTO.它不需要知道如何分割输入数据,或者如何聚合多个步骤执行的结果。一般来说它可能也不需要了解弹性或故障转移,因为在许多情况下这些都是结构的特性,无论如何Spring Batch总是提供了独立于结构的可重启能力: 一个失败的作业总是会被重新启动,并且只会重新执行失败的步骤。

**PartitionHandler**接口可以有各种结构的实现类: 如简单RMI远程方法调用,EJB远程调用,自定义web服务、JMS、Java Spaces, 共享内存网格(如Terracotta或Coherence)、网格执行结构(如GridGain)。Spring Batch自身不包含任何专有网格或远

程结构的实现。

但是 Spring Batch也提供了一个有用的**PartitionHandler**实现，在本地分开的线程中执行Steps,该实现类名为**TaskExecutorPartitionHandler**,并且他是上面的XML配置中的默认处理器。还可以像下面这样明确地指定:

```
<step id="step1.master">
    <partition step="step1" handler="handler"/>
</step>

<bean class="org.spr...TaskExecutorPartitionHandler">
    <property name="taskExecutor" ref="taskExecutor"/>
    <property name="step" ref="step1" />
    <property name="gridSize" value="10" />
</bean>
```

**gridSize**决定要创建的独立的step执行的数量,所以它可以配置为**TaskExecutor**中线程池的大小,或者也可以设置得比可用的线程数稍大一点,在这种情况下,执行块变得更小一些。

**TaskExecutorPartitionHandler** 对于IO密集型步骤非常给力,比如要拷贝大量的文件,或复制文件系统到内容管理系统时。它还可用于远程执行的实现,通过为远程调用提供一个代理的步骤实现(例如使用Spring Remoting)。

## 7.4.2 分割器(Partitioner)

分割器有一个简单的职责: 仅为新的step实例生成执行环境(contexts),作为输入参数(这样重启时就不需要考虑)。 该接口只有一个方法:

```
public interface Partitioner {
    Map<String, ExecutionContext> partition(int gridSize);
}
```

这个方法的返回值是一个Map对象,将每个Step执行分配的唯一名称(Map泛型中的 **String**),和与其相关的输入参数以 **ExecutionContext** 的形式做一个映射。 这个名称随后在批处理 meta data 中作为分区 **StepExecutions** 的Step名字显示。 **ExecutionContext**仅仅只是一些 名-值对的集合,所以它可以包含一系列的主键,或行号,或者是输入文件的位置。 然后远程**Step** 通常使用 **#{...}**占位符来绑定到上下文输入(在 step作用域内的后期绑定),详情请参见下一节。

step执行的名称( **Partitioner**接口返回的 **Map** 中的 key)在整个作业的执行过程中需要保持唯一,除此之外没有其他具体要求。 要做到这一点,并且需要一个对用户有意义的名称,最简单的方法是使用 前缀+后缀 的命名约定,前缀可以是被执行的Step的名称(这本身在作业**Job**中就是唯一的),后缀可以是一个计数器。在框架中有一个使用此约定的 **SimplePartitioner**。

有一个可选接口 **PartitioneNameProvider** 可用于和分区本身独立的提供分区名称。 如果一个 **Partitioner** 实现了这个接口, 那么重启时只有names会被查询。 如果分区是重量级的，那么这可能是一个很有用的优化。 很显然,**PartitioneNameProvider**提供的名称必须和**Partitioner**提供的名称一致。

## 7.4.3 将输入数据绑定到 Steps

因为step的输入参数在运行时绑定到ExecutionContext中,所以由相同配置的PartitionHandler执行的steps是非常高效的。 通过 Spring Batch的StepScope特性这很容易实现(详情请参考 后期绑定)。 例如,如果 **Partitioner** 创建 **ExecutionContext** 实例, 每个step执行都以**fileName**为key 指向另一个不同的文件(或目录),则 **Partitioner** 的输出看起来可能像下面这样:

表 **7.1.** 由执行目的目录处理**Partitioner**提供的的**step**执行上下文名称示例

```
| Step Execution Name (key) |    ExecutionContext (value)  |
| filecopy:partition0        |   fileName=/home/data/one    |
```

```
| filecopy:partition1        |   fileName=/home/data/two   |
| filecopy:partition2        |   fileName=/home/data/three |
```

然后就可以将文件名绑定到 step 中, step使用了执行上下文的后期绑定:

```
<bean id="itemReader" scope="step"
      class="org.spr...MultiResourceItemReader">
    <property name="resource" value="#{stepExecutionContext[fileName]}/*"/>
</bean>
```

# 重复执行

# 重试处理

重试处理

# 单元测试

单元测试

# 通用批处理模式

# 12. JSR-352 支持

Spring Batch 3.0 对 JSR-352 提供完整的支持. 本节并不讲述这个规范, 而是讲解如何将 JSR-352 的相关概念应用于Spring Batch. 关于JSR-352 的更多信息可以参考 JCP 网站: https://jcp.org/en/jsr/detail?id=352

# Spring Batch Integration模块

# 附录A ItemReader 与 ItemWriter 列表

## A.1 Item Readers

**Table A.1.** 所有可用的**Item Reader**列表

| Item Reader | 说明 |
| --- | --- |
| AbstractItemCountingItemStreamItemReader | 抽象基类，支持重启，通过统计(counting)从 `ItemReader` 返回对象的数量来实现. |
| AggregateItemReader | 此 ItemReader 提供一个 list , 用来存储 ItemReader 读取的对象, 直到他们已准备装配为一个集合。<br>此 ItemReader 通过 FieldSetMapper 的常量值 AggregateItemReader#**BEGIN_RECORD** 以及 AggregateItemReader#**END_RECORD** 来标记记录的开始与结束。 |
| AmqpItemReader | 给定一个提供同步获取方法( synchronous receive methods)的 Spring AmqpTemplate. 使用 receiveAndConvert() 方法可以得到 POJO 对象. |
| FlatFileItemReader | 从平面文件(flat file)中读取数据，支持 ItemStream 以及 Skippable 特性. 请参考 Read from a File 一节 |
| HibernateCursorItemReader | 从基于 HQL 查询的 cursor 中读取数据。 请参考 Reading from a Database 一节。 |
| HibernatePagingItemReader | 从分页(paginated)的HQL查询中读取数据 |
| IbatisPagingItemReader | 通过 iBATIS 的分页查询读取数据， 对于大型数据集,分页能避免内存不足/溢出的问题. 请参考: HOWTO - Read from a Database. 这个 ItemReader 在 Spring Batch 3.0 中已废弃. |
| ItemReaderAdapter | 将任意类适配到 `ItemReader` 接口. |
| JdbcCursorItemReader | 通过 JDBC 从 一个 database cursor 中读取数据. 请参考: HOWTO - Read from a Database |
| JdbcPagingItemReader | 给定一个 SQL statement, 通过分页查询读取数据， 避免读取大型数据集时 内存不足/溢出的问题 |
| JmsItemReader | 给一个 Spring JmsOperations 对象和一个 JMS Destination 对象/也可以是用来发送错误的 destination name , 调用注入的 JmsOperations 里面的 receive() 方法来获取对象 |
| JpaPagingItemReader | 给定一个 JPQL statement, 通过分页查询读取数据， 避免读取大型数据集时 内存不足/溢出的问题 |
| ListItemReader | 从 list 中读取数据， 一次返回一条 |
| MongoItemReader | 给定一个 MongoOperations 对象,以及从 MongoDB 中查询数据所使用的JSON, 通过 MongoOperations 的 find 方法来获取数据 |
| Neo4jItemReader | 给定一个 Neo4jOperations 对象,以及一个 Cyhper query 所需的 components, 将 Neo4jOperations.query 方法的结果返回 |
| RepositoryItemReader | 给定一个 Spring Data PagingAndSortingRepository 对象, 一个 Sort 对象,以及要执行的 method name, 返回 Spring Data repository 实现提供的数据 |
| StoredProcedureItemReader | 执行存储过程(database stored procedure),从返回的 database cursor 中读取数据. 请参考: HOWTO - Read from a Database |
| StaxEventItemReader | 通过 StAX 读取. 请参考 HOWTO - Read from a File |

# A.2 Item Writers

**Table A.2.** 所有可用的 **Item Writer** 列表

| **Item Writer** | 说明 |
| --- | --- |
| AbstractItemStreamItemWriter | 抽象基类, 组合了 `ItemStream` 和 `ItemWriter` 接口。 |
| AmqpItemWriter | 给定一个提供同步发送方法的 Spring AmqpTemplate. 使用 convertAndSend(Object) 方法可以输出 POJO 对象. |
| CompositeItemWriter | 将注入的 **List** 里面每一个元素都传给 **ItemWriter** 的处理方法 |
| FlatFileItemWriter | 写入平面文件(flat file). 支持 ItemStream 以及 Skippable 特性. 请参考 Writing to a File 一节 |
| GemfireItemWriter | 使用 GemfireOperations 对象, 根据配置的 delete 标志, 对 items 进行写入或者删除 |
| HibernateItemWriter | 这个 item writer 是Hibernate会话相关的(hibernate session aware), 用来处理非 hibernate 相关的组件(non-"hibernate aware" )不需要关心的事务性工作, 并且委托另一个 item writer 来执行实际的写入工作. |
| IbatisBatchItemWriter | 在批处理中直接使用 iBatis 的 API. 这个 ItemWriter 在 Spring Batch 3.0 中已废弃. |
| ItemWriterAdapter | 将任意类适配到 `ItemWriter` 接口. |
| JdbcBatchItemWriter | 尽可能地利用 `PreparedStatement` 的批处理功能(batching features), 还可以采取基本的步骤来定位 `flush` 失败等问题。 |
| JmsItemWriter | 利用 JmsOperations 对象, 通过 JmsOperations.convertAndSend() 方法将 items 写入到默认队列( default queue) |
| JpaItemWriter | 这个 item writer 是 JPA EntityManager aware 的, 用来处理非jpa相关的(non-"jpa aware") `ItemWriter` 不需要关心的事务性工作, 并且委托另一个 item writer 来执行实际的写入工作. |
| MimeMessageItemWriter | 通过 Spring 的 JavaMailSender 对象, 类型为 `MimeMessage` 的 item 可以作为 mail messages 发送出去 |
| MongoItemWriter | 给定一个 MongoOperations 对象, 数据通过 MongoOperations.save(Object) 方法写入. 实际的写操作会推迟到事务提交时才执行. |
| Neo4jItemWriter | 给定一个 Neo4jOperations 对象, item 通过 save(Object) 方法完成持久化, 或者通过 delete(Object) 方法来删除, 取决于 `ItemWriter` 的配置 |
| PropertyExtractingDelegatingItemWriter | 扩展 AbstractMethodInvokingDelegator 创建动态参数. 动态参数是通过注入的 field name数组,从(SpringBeanWrapper)处理的 item 中获取的 |
| RepositoryItemWriter | 给定一个 Spring Data CrudRepository 实现, 则使用配置文件指定的方法保存 item。 |
| StaxEventItemWriter | 通过**ObjectToXmlSerializer** 对象将每个 item 转换为XML, 然后用 StAX 将这些内容写入 XML 文件。 |

# 附录B

# 附录C

# Appendix C. Batch Processing and Transactions

## C.1 Simple Batching with No Retry

Consider the following simple example of a nested batch with no retries. This is a very common scenario for batch processing, where an input source is processed until exhausted, but we commit periodically at the end of a "chunk" of processing.

```
1   |  REPEAT(until=exhausted) {
|
2   |    TX {
3   |      REPEAT(size=5) {
3.1 |        input;
3.2 |        output;
|      }
|    }
|
|  }
```

The input operation (3.1) could be a message-based receive (e.g. JMS), or a file-based read, but to recover and continue processing with a chance of completing the whole job, it must be transactional. The same applies to the operation at (3.2) - it must be either transactional or idempotent.

If the chunk at REPEAT(3) fails because of a database exception at (3.2), then TX(2) will roll back the whole chunk.

## C.2 Simple Stateless Retry

It is also useful to use a retry for an operation which is not transactional, like a call to a web-service or other remote resource. For example:

```
0   |  TX {
1   |    input;
1.1 |    output;
2   |    RETRY {
2.1 |      remote access;
|    }
|  }
```

This is actually one of the most useful applications of a retry, since a remote call is much more likely to fail and be retryable than a database update. As long as the remote access (2.1) eventually succeeds, the transaction TX(0) will commit. If the remote access (2.1) eventually fails, then the transaction TX(0) is guaranteed to roll back.

## C.3 Typical Repeat-Retry Pattern

The most typical batch processing pattern is to add a retry to the inner block of the chunk in the Simple Batching example. Consider this:

```
1   |  REPEAT(until=exhausted, exception=not critical) {
```

```
|
2   |    TX {
3   |      REPEAT(size=5) {
|
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
5.1 |          output;
6   |        } SKIP and RECOVER {
|          notify;
|        }
|
|      }
|    }
|
|  }
```

The inner RETRY(4) block is marked as "stateful" - see the typical use case for a description of a stateful retry. This means that if the the retry PROCESS(5) block fails, the behaviour of the RETRY(4) is as follows.

- Throw an exception, rolling back the transaction TX(2) at the chunk level, and allowing the item to be re-presented to the input queue.
- When the item re-appears, it might be retried depending on the retry policy in place, executing PROCESS(5) again. The second and subsequent attempts might fail again and rethrow the exception.
- Eventually the item re-appears for the final time: the retry policy disallows another attempt, so PROCESS(5) is never executed. In this case we follow a RECOVER(6) path, effectively "skipping" the item that was received and is being processed.

Notice that the notation used for the RETRY(4) in the plan above shows explictly that the the input step (4.1) is part of the retry. It also makes clear that there are two alternate paths for processing: the normal case is denoted by PROCESS(5), and the recovery path is a separate block, RECOVER(6). The two alternate paths are completely distinct: only one is ever taken in normal circumstances.

In special cases (e.g. a special TranscationValidException type), the retry policy might be able to determine that the RECOVER(6) path can be taken on the last attempt after PROCESS(5) has just failed, instead of waiting for the item to be re-presented. This is not the default behavior because it requires detailed knowledge of what has happened inside the PROCESS(5) block, which is not usually available - e.g. if the output included write access before the failure, then the exception should be rethrown to ensure transactional integrity.

The completion policy in the outer, REPEAT(1) is crucial to the success of the above plan. If the output(5.1) fails it may throw an exception (it usually does, as described), in which case the transaction TX(2) fails and the exception could propagate up through the outer batch REPEAT(1). We do not want the whole batch to stop because the RETRY(4) might still be successful if we try again, so we add the exception=not critical to the outer REPEAT(1).

Note, however, that if the TX(2) fails and we do try again, by virtue of the outer completion policy, the item that is next processed in the inner REPEAT(3) is not guaranteed to be the one that just failed. It might well be, but it depends on the implementation of the input(4.1). Thus the output(5.1) might fail again, on a new item, or on the old one. The client of the batch should not assume that each RETRY(4) attempt is going to process the same items as the last one that failed. E.g. if the termination policy for REPEAT(1) is to fail after 10 attempts, it will fail after 10 consecutive attempts, but not necessarily at the same item. This is consistent with the overall retry strategy: it is the inner RETRY(4) that is aware of the history of each item, and can decide whether or not to have another attempt at it.

C.4 Asynchronous Chunk Processing The inner batches or chunks in the typical example above can be executed concurrently by configuring the outer batch to use an AsyncTaskExecutor. The outer batch waits for all the chunks to complete before completing.

```
1   |  REPEAT(until=exhausted, concurrent, exception=not critical) {
|
2   |    TX {
```

```
3   |      REPEAT(size=5) {
|
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
|          output;
6   |        } RECOVER {
|          recover;
|        }
|
|      }
|    }
|
|  }
```

C.5 Asynchronous Item Processing The individual items in chunks in the typical can also in principle be processed concurrently. In this case the transaction boundary has to move to the level of the individual item, so that each transaction is on a single thread:

```
1   |  REPEAT(until=exhausted, exception=not critical) {
|
2   |    REPEAT(size=5, concurrent) {
|
3   |      TX {
4   |        RETRY(stateful, exception=deadlock loser) {
4.1 |          input;
5   |        } PROCESS {
|          output;
6   |        } RECOVER {
|          recover;
|        }
|      }
|
|    }
|
|  }
```

This plan sacrifices the optimisation benefit, that the simple plan had, of having all the transactional resources chunked together. It is only useful if the cost of the processing (5) is much higher than the cost of transaction management (3).

C.6 Interactions Between Batching and Transaction Propagation There is a tighter coupling between batch-retry and TX management than we would ideally like. In particular a stateless retry cannot be used to retry database operations with a transaction manager that doesn't support NESTED propagation.

For a simple example using retry without repeat, consider this:

```
1   |  TX {
|
1.1 |    input;
2.2 |    database access;
2   |    RETRY {
3   |      TX {
3.1 |        database access;
|      }
|    }
|
|  }
```

Again, and for the same reason, the inner transaction TX(3) can cause the outer transaction TX(1) to fail, even if the RETRY(2) is eventually successful.

Unfortunately the same effect percolates from the retry block up to the surrounding repeat batch if there is one:

```
1   |  TX {
|
2   |    REPEAT(size=5) {
2.1 |      input;
2.2 |      database access;
3   |      RETRY {
4   |        TX {
4.1 |          database access;
|        }
|      }
|    }
|
|  }
```

Now if TX(3) rolls back it can pollute the whole batch at TX(1) and force it to roll back at the end.

What about non-default propagation?

- In the last example PROPAGATION_REQUIRES_NEW at TX(3) will prevent the outer TX(1) from being polluted if both transactions are eventually successful. But if TX(3) commits and TX(1) rolls back, then TX(3) stays committed, so we violate the transaction contract for TX(1). If TX(3) rolls back, TX(1) does not necessarily (but it probably will in practice because the retry will throw a roll back exception).
- PROPAGATION_NESTED at TX(3) works as we require in the retry case (and for a batch with skips): TX(3) can commit, but subsequently be rolled back by the outer transaction TX(1). If TX(3) rolls back, again TX(1) will roll back in practice. This option is only available on some platforms, e.g. not Hibernate or JTA, but it is the only one that works consistently.

So NESTED is best if the retry block contains any database access.

C.7 Special Case: Transactions with Orthogonal Resources Default propagation is always OK for simple cases where there are no nested database transactions. Consider this (where the SESSION and TX are not global XA resources, so their resources are orthogonal):

```
0   |  SESSION {
1   |    input;
2   |    RETRY {
3   |      TX {
3.1 |        database access;
|      }
|    }
|  }
```

Here there is a transactional message SESSION(0), but it doesn't participate in other transactions with PlatformTransactionManager, so doesn't propagate when TX(3) starts. There is no database access outside the RETRY(2) block. If TX(3) fails and then eventually succeeds on a retry, SESSION(0) can commit (it can do this independent of a TX block). This is similar to the vanilla "best-efforts-one-phase-commit" scenario - the worst that can happen is a duplicate message when the RETRY(2) succeeds and the SESSION(0) cannot commit, e.g. because the message system is unavailable.

C.8 Stateless Retry Cannot Recover The distinction between a stateless and a stateful retry in the typical example above is important. It is actually ultimately a transactional constraint that forces the distinction, and this constraint also makes it obvious why the distinction exists.

We start with the observation that there is no way to skip an item that failed and successfully commit the rest of the chunk unless we wrap the item processing in a transaction. So we simplify the typical batch execution plan to look like this:

```
0   |  REPEAT(until=exhausted) {
|
```

```
1   |    TX {
2   |      REPEAT(size=5) {
|
3   |        RETRY(stateless) {
4   |          TX {
4.1 |            input;
4.2 |            database access;
|          }
5   |        } RECOVER {
5.1 |          skip;
|        }
|
|      }
|    }
|
|  }
```

Here we have a stateless RETRY(3) with a RECOVER(5) path that kicks in after the final attempt fails. The "stateless" label just means that the block will be repeated without rethrowing any exception up to some limit. This will only work if the transaction TX(4) has propagation NESTED.

If the TX(3) has default propagation properties and it rolls back, it will pollute the outer TX(1). The inner transaction is assumed by the transaction manager to have corrupted the transactional resource, and so it cannot be used again.

Support for NESTED propagation is sufficiently rare that we choose not to support recovery with stateless retries in current versions of Spring Batch. The same effect can always be achieved (at the expense of repeating more processing) using the typical pattern above.

# 术语表(Glossary)

## Spring Batch术语表

### Batch 批

An accumulation of business transactions over time.

随着时间累积而形成的一批业务事务。

### Batch Application Style (批处理程序风格)

Term used to designate batch as an application style in its own right similar to online, Web or SOA. It has standard elements of input, validation, transformation of information to business model, business processing and output. In addition, it requires monitoring at a macro level.

用来称呼批处理自身的程序风格的术语，类似于 online, Web 或者 SOA。 其具有的标准元素包括： 输入、验证、将信息转换为业务模型、 业务处理 以及 输出。此外,还需要在宏观层面上进行监控。

### Batch Processing (批处理任务)

The handling of a batch of many business transactions that have accumulated over a period of time (e.g. an hour, day, week, month, or year). It is the application of a process, or set of processes, to many data entities or objects in a repetitive and predictable fashion with either no manual element, or a separate manual element for error processing.

积累了一定时间周期(比如小时、天、周、月或年)的业务事务归到一批进行处理。这种程序可以有一个进程,或者一组进程; 以重复可预测的方式处理很多数据实体/对象,要么没有人工干预，或者有些错误需要人工单独进行处理。

### Batch Window (批处理窗口)

The time frame within which a batch job must complete. This can be constrained by other systems coming online, other dependent jobs needing to execute or other factors specific to the batch environment.

批处理作业必须在这个时间范围内完成。 可能受到的制约包括: 其他系统要上线, 相关作业要执行, 或者是特定于批处理环境的其他因素 。

### Step (步骤)

It is the main batch task or unit of work controller. It initializes the business logic, and controls the transaction environment based on commit interval setting, etc.

这是主要的批处理任务, 或者工作控制器的组成单元。在其中进行业务逻辑初始化, 基于提交间隔控制事务环境,等等。

### Tasklet (小任务)

A component created by application developer to process the business logic for a Step.

由程序员创建的组件, 用来处理某个 step 中的业务逻辑。

### Batch Job Type (批处理作业类型)

Job Types describe application of jobs for particular type of processing. Common areas are interface processing (typically flat files), forms processing (either for online pdf generation or print formats), report processing.

作业类型描述特定类型的作业处理程序。共同领域包括 接口处理(通常是平面文件), 格式处理(如在线生成pdf或打印格式), 报表处理等。

## Driving Query (驱动查询)

A driving query identifies the set of work for a job to do; the job then breaks that work into individual units of work. For instance, identify all financial transactions that have a status of "pending transmission" and send them to our partner system. The driving query returns a set of record IDs to process; each record ID then becomes a unit of work. A driving query may involve a join (if the criteria for selection falls across two or more tables) or it may work with a single table.

一次驱动查询用来标识一个作业要做的工作组; 然后工作被打散为单个的工作单元。例如, 找出所有状态为"等待传输"的金融交易 并发送给合作伙伴系统。驱动查询返回要处理的记录的ID集合; 每个记录ID 稍后都会成为一个工作单元。 一次驱动查询可能涉及 join连接(如果条件遇到两个或多个表), 也可能只使用单个表。

## Item (数据项)

An item represents the smallest ammount of complete data for processing. In the simplest terms, this might mean a line in a file, a row in a database table, or a particular element in an XML file.

一个 item 代表要处理的最小的完整的数据。 最简单的理解, 可以是文件中的一行(line), 数据表中的一行(row), 或者XML文件中一个特定的元素(element)。

## Logical Unit of Work (LUW, 逻辑工作单元)

> 原文可能错了, `Logicial`

A batch job iterates through a driving query (or another input source such as a file) to perform the set of work that the job must accomplish. Each iteration of work performed is a unit of work.

批处理作业通过驱动查询(或者是文件之类的输入源)来迭代执行必须完成的工作。工作执行中的每次迭代就是一个工作单元。

## Commit Interval (提交区间)

A set of LUWs processed within a single transaction.

在单个事务中处理的 逻辑工作单元集合.

## Partitioning (分块, 分区)

Splitting a job into multiple threads where each thread is responsible for a subset of the overall data to be processed. The threads of execution may be within the same JVM or they may span JVMs in a clustered environment that supports workload balancing.

将一个作业拆分给多个线程来执行, 每个线程只负责处理整个数据中的一部分。这些线程可能在同一个JVM中执行, 也可能跨越JVM在支持工作负载平衡的集群环境中运行。

## Staging Table (分段表,阶段表)

A table that holds temporary data while it is being processed.

一个存储临时数据的表, 里面的数据即将被处理。

## Restartable (可再次启动的)

A job that can be executed again and will assume the same identity as when run initially. In otherwords, it is has the same job instance id.

可再次执行的作业, 而且再次执行时, 和初次运行具有同样的身份。换言之, 两者具有相同的作业实例 id.

## Rerunnable (可再次运行)

A job that is restartable and manages its own state in terms of previous run's record processing. An example of a rerunnable step is one based on a driving query. If the driving query can be formed so that it will limit the processed rows when the job is restarted than it is re-runnable. This is managed by the application logic. Often times a condition is added to the where statement to limit the rows returned by the driving query with something like "and processedFlag != true".

可再次启动的作业, 并且可根据之前的处理记录来合理调整自身的状态。可再次运行 Step 的一个例子是基于 driving query 的部分。如果是 re-runnable 的, 那么当 restarted 后, driving query 就会排除已经处理过的那些行。当然这由应用程序逻辑决定。通常是在 where 子句中添加条件来限制查询返回的结果, 例如 “processedFlag != true”。

## Repeat (重复)

One of the most basic units of batch processing, that defines repeatability calling a portion of code until it is finished, and while there is no error. Typically a batch process would be repeatable as long as there is input.

批处理最基本的单元之一, 定义了可重复调用的一部分代码, 直到完成某个任务为止, 如果不出错的话。通常来说, 只要还有输入数据, 批处理过程就会一直重复。

## Retry (重试)

Simplifies the execution of operations with retry semantics most frequently associated with handling transactional output exceptions. Retry is slightly different from repeat, rather than continually calling a block of code, retry is stateful, and continually calls the same block of code with the same input, until it either succeeds, or some type of retry limit has been exceeded. It is only generally useful if a subsequent invocation of the operation might succeed because something in the environment has improved.

简化了的重试语义通常和事务输出异常处理有关。重试(Retry)和重复(Repeat)略有不同, 不仅仅是持续不断地调用某个代码块, 因为重试是有状态的, 所以每次都是使用相同的输入, 直到成功为止, 或者是已经超过了某种类型的重试限制。一般只有在依赖某种外部环境的情况下, 如果外部环境得到改善, 就会使得后续操作会成功的情况下就会很有用。

## Recover (恢复)

Recover operations handle an exception in such a way that a repeat process is able to continue.

恢复操作用来对付异常, 通过这种方式使重复过程得以继续下去。

## Skip (跳过)

Skip is a recovery strategy often used on file input sources as the strategy for ignoring bad input records that failed validation.

跳过是一种容错策略, 通常在读取文件输入时，用来忽略验证失败的脏数据。